# An Approach to Concurrent Semantics Using Complete Traces

Kevin S. Van Horn

Computer Science Department
California Institute of Technology

5236:TR:86

| Report Documentation Page | *Form Approved* *OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| 1. REPORT DATE **15 DEC 1986** | 2. REPORT TYPE | 3. DATES COVERED **00-00-1986 to 00-00-1986** |
| 4. TITLE AND SUBTITLE **An Approach to Concurrent Semantics Using Complete Traces** | | 5a. CONTRACT NUMBER |
| | | 5b. GRANT NUMBER |
| | | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | | 5d. PROJECT NUMBER |
| | | 5e. TASK NUMBER |
| | | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714** | | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | | 10. SPONSOR/MONITOR'S ACRONYM(S) |
| | | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |
| 12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited** | | |
| 13. SUPPLEMENTARY NOTES | | |
| 14. ABSTRACT **see report** | | |
| 15. SUBJECT TERMS | | |

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT **unclassified** | b. ABSTRACT **unclassified** | c. THIS PAGE **unclassified** | | **59** | |

**Standard Form 298 (Rev. 8-98)**
Prescribed by ANSI Std Z39-18

# An Approach to Concurrent Semantics Using Complete Traces

Kevin S. Van Horn[1]

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

5236:TR:86

Computer Science 256-80
California Institute of Technology
Pasadena, CA 91125

December 15, 1986

[1]Work supported in part by the Defense Advanced Research Projects Agency, ARPA Order No. 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597

## Acknowledgements

Thanks are due to my advisor, Alain J. Martin, whose criticisms have been invaluable, and also to Young-il Choo, whose remarks have several times given me new perspectives on my work.

# Contents

Chapter 1

# Introduction

In this thesis we address the problem of how to give acceptable formal semantics for concurrent programming languages, that is, how to give a precise mathematical formulation of what programs in various such languages mean. We take a broad view of what constitutes a programming language — for example, notations that describe digital circuits or Petri nets we consider programming languages, as well as notations that describe systems of concurrent processes interacting via shared variables or message-passing. It is important that we have a formal semantics for any programming language we use, for the following reasons:

1. It is needed in order to precisely define the language.
2. It is needed to provide some basis for deciding the correctness of implementations of the language.
3. It is needed to provide a foundation for methods of analyzing and reasoning about the behavior of programs in the language.

In this thesis we shall not concern ourselves with 2. and 3. above, but shall concentrate on the problem of specifying the semantics of concurrent programming languages and showing the semantics to be 'acceptable' in some sense.

## 1.1 What kind of semantics?

In order to be considered 'acceptable', a formal semantics for a programming language should satisfy the following:

1. It should formalize, in some way, our intuitive notions of what a program means.
2. It should be appropriately abstract, i.e. it should distinguish between two programs if and only if we would feel them to differ in some important manner.
3. It should be capable of supporting modular reasoning about programs. This means that it should be possible to prove properties of a program (or subprogram) by breaking it into parts and separately proving properties of these parts.

We will discuss one more criterion for acceptability later.

There are at present three major approaches to programming language semantics: *operational* semantics, *axiomatic* semantics, and *denotational* semantics. Loosely speaking, an operational semantics defines a machine that executes the program. Since an operational semantics will generally satisfy neither criterion 2. nor 3., we forego this approach. An axiomatic semantics is a logical system that is used to prove properties of programs. A denotational semantics is a *semantic function* that assigns to any program in the language some mathematical object, called the program's denotation; this function is defined by recursion on the syntactic structure of the program, and thus satisfies criterion 3.

As is to be expected, an axiomatic semantics is usually easier to use than a denotational semantics for proving properties of programs, since it includes a language for expressing program properties that is designed to be as convenient as possible. A denotational semantics, however, can often give more insight into the semantic issues of a language. It can also serve as a foundation for an axiomatic semantics, justifying the axioms and proof rules and providing a ready-made proof of the consistency of the axiom set used.

The approach we will take is essentially a denotational one: we give the semantics of a programming language in terms of a semantic function. However, it bears some resemblance to the axiomatic approach in that we will make heavy use of the language of temporal logic [17,4] in defining our semantic functions. We will also depart from a strictly denotational approach in that we may define our semantic functions using syntactic transformations on programs or subprograms, and not simply by recursion on the syntactic structure of a program.

**Example.** Since we have chosen a denotational approach, we give a simple (and much-used) example of a denotational semantics. The language is that of arithmetic expressions, given by the following grammar:

$$E ::= N \mid E+E \mid E-E \mid E*E \mid E/E \mid \mathbf{ifz}\ E\ \mathbf{then}\ E\ \mathbf{else}\ E$$

($N$ is the set of numerals.) We give the denotational semantics of this language as the function $\mathcal{E}: E \to \mathbf{Z}$ defined by

$$\begin{aligned} &\mathcal{E}[\![\overline{n}]\!] \equiv n \qquad \text{where } \overline{n} \text{ is the numeral representing the integer } n \\ &\mathcal{E}[\![e_1+e_2]\!] \equiv \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!] \\ &\quad\vdots \\ &\mathcal{E}[\![\mathbf{ifz}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!] \equiv \begin{cases} \mathcal{E}[\![e_1]\!] & \text{if } \mathcal{E}[\![e]\!] = 0 \\ \mathcal{E}[\![e_2]\!] & \text{if } \mathcal{E}[\![e]\!] \neq 0 \end{cases} \end{aligned}$$

## 1.2 What is a concurrent program?

Having decided upon an essentially denotational approach, we must now decide what kind of an object a concurrent program should denote. Implicit in the previous statement is that, in this thesis, we propose a uniform approach to the semantics of concurrent programming languages — we wish programs in ***any*** concurrent language to denote the same kind of object. The reason for this is a desire for conceptual economy and generality; we do not want to have to start from scratch every time we look at a new language. In addition, such a uniformity is bound to facilitate the introduction of useful new programming constructs into a language without wreaking havoc with the semantics. An example of how an insufficiently general approach can cause problems in this regard is the various denotational semantics of CSP [12] proposed in [7,11,24]; there is no evident way of extending these to include the probe function of A. J. Martin [18], which is a very useful extension to CSP.

We note that a common notion of concurrent languages is that their programs may be viewed as describing a system of (concurrently) interacting objects, which we shall henceforth call *agents.*

This may in fact be taken as an informal definition of what we mean by a concurrent language. For a digital circuit the agents will be logic gates, flip-flops, etc.; for a CSP program the agents will be the various communicating processes. The denotation of a concurrent program should then describe the behavior of the indicated system.

Physical systems are generally described by some set of time-varying quantities, which we shall call the *state* of the system. Formally, there is some function $f$ such that $f(t)$ is the state of the system at time t. $f$ is usually not given directly; instead, a set of constraints on $f$ are given (e.g., differential equations involving $f$, boundary conditions, etc.), and these are used to derive $f$ explicitly or to deduce additional information about $f$. As examples, if the system in question is an electrical circuit then, for all t, $f(t)$ is a vector of voltages and current flows at various points in the circuit, and if the system is an oscillating string then, for all t, $f(t)$ is a continuous function giving the displacement of each point on the string.

The systems we are interested in may be considered *discrete* abstractions of physical systems. By 'discrete' we mean that any finite interval of time may be partitioned into a finite number of sub-intervals over which the system state is constant. Formally, there is some denumerable set $T = \{t_0, t_1, \ldots\}$ of real numbers such that

1. $t_0 = 0$ and $t_i < t_{i+1}$ for all $i$;
2. for all t there exists some $i$ s.t. $t_i > t$;
3. $f(t) = f(t_i)$ for all $t_i \leq t < t_{i+1}$.

Condition 2. guarantees that any finite interval of time contains only a finite number of elements of T, and 3. says that the state may change only at times in T. Note that all T satisfying 1–3 have a common subset, consisting of 0 and the times at which the state changes.

For any such T we define the infinite sequence

$$\sigma(f, T) \equiv f(t_0)\, f(t_1)\, f(t_2) \cdots.$$

(We write '$\equiv$' for 'is defined to be'.) If we are not interested in knowing the times at which the state changes, then $\sigma(f, T)$, which we call the *complete trace* of the system with respect to T, fully describes the behavior of the system. We define a complete trace of the system to be any element of

$$\{\, \sigma(f, T) \mid T \text{ satisfies conditions 1–3} \,\}.$$

It is easily verified that this set is always nonempty. The complete traces of a system are all equivalent in the sense that they differ only in the number of consecutive instances of the same state.

In most cases the description of a system's behavior will not uniquely determine $f$. For example, in giving the semantics of a notation intended to describe asynchronous digital circuits we might wish to make no (or limited) assumptions about the relative speeds of the components, and no assumptions about how the values on the input lines will change. The denotation of a program in this notation should then be a *set* of traces (sequences), with the interpretation that, whatever the relative speeds of the components, whatever the behavior of the input lines, etc., any complete trace of the system described is in this set.

In general then, a concurrent program will denote a set of infinite traces, with the interpretation that any complete trace of the system described is in this set. Equivalently, a concurrent program denotes a *predicate* on traces that is satisfied by any complete trace of the system.

## 1.3 Statement of intent

The above leads us to one final criterion of acceptability for any formal semantics of a concurrent language: the denotation of any program should be a *nonempty* trace set. Let $T$ be the trace set denoted by some program $P$. There would be severe consequences if we allowed $T$ to be empty; in particular, in reasoning about the system described by our program we would begin with the false premiss '$t \in T$' ($t$ being an arbitrary complete trace of the system), and hence find ourselves miraculously capable of proving any statement whatsoever. If the semantics we give for a concurrent language is to be used to justify some proof system for it, it is necessary that the trace set denoted by any program be nonempty for the proof system to be sound.

We may now state more precisely the aim of this thesis. It is twofold:

1. To present an approach to the semantics of concurrent programming languages in which the denotation of a program is always a nonempty set of traces, and to give such a semantics for several concurrent programming languages.

2. To show how it may be guaranteed that the trace set denoted by a concurrent program is nonempty.

## 1.4 Comparison with other work

The approach to concurrent semantics presented herein differs from other approaches that use traces, such as Milner's CCS [19], the failures model and other models of CSP [12,7,11] and trace theory [23,22], in a number of ways:

1. We consider the notion of a system's state to be more fundamental than the notion of an action; hence we use sequences of states instead of sequences of actions.

2. There is no synchronization of actions performed by distinct agents. As we shall see in Chapter 4, this is not necessary to describe the tight synchronization used in languages such as CSP.

3. We focus on complete traces of a system instead of partial traces, i.e. traces describing the system's evolution up to some arbitrary but finite point in time. Many interesting liveness properties, such as fairness, cannot be adequately expressed in terms of partial traces [5].

4. We do not shy away from infinities. Rather than limit ourselves to finite traces constructed from a finite alphabet, we deal with the consequences of allowing having traces and infinite alphabets. Neither do we impose restrictive closure requirements on our sets of traces, as in [24,1].

This work is most closely related to the work in linear-time temporal logic [17,2,13,3], which we shall simply refer to as 'temporal logic'. Temporal logic is a logic for reasoning about sequences of states, and its language is a convenient one for expressing many predicates on such sequences.

The early work in temporal logic (as in [17]) gave the semantics of a program as a set of temporal logic axioms. These axioms were justified by giving an operational semantics that produced, for any program, a set of traces, all of which satisfied the axioms corresponding to the program. Unfortunately, the semantics was non-compositional. More recent work [2,13,3] has focused on giving compositional, axiomatic semantics using temporal logic. However, these efforts have not adequately addressed the issue of the consistency of their axiom sets. Lamport in [13] states that the axioms corresponding to a program in his approach may indeed be inconsistent, but he considers this to be unimportant. Barringer, Kuiper and Pnueli in [2] justify their proof rule for recursive procedure calls by stating that the semantics of a recursive procedure definition is the maximal fixed-point of a certain temporal logic formula, but do not prove that there exists any trace satisfying this maximal fixed-point. A similar comment applies to [3].

# Chapter 2

# Predicates on Traces

In this chapter we present a notation for expressing predicates on traces. It is a variant of the past-time temporal logic notations presented in [15,4]. We shall use this notation throughout the thesis, and in particular we shall use it in this chapter to give the semantics of a very simple programming language, similar to the one use by Chandy and Misra in [6].

## 2.1 Traces

First of all we define the operations and predicates we shall use in discussing traces. A *trace* is a sequence of elements taken from some nonempty set. $Q^+$ is the set of all finite traces of nonzero length formed from (nonempty) set $Q$, $Q^\omega$ is the set of all infinite traces formed from $Q$, and $Q^\infty$ is $Q^+ \cup Q^\omega$. $\varepsilon$ is the unique trace of zero length.

**Definition:** Given $v \in Q^+ \cup \{\varepsilon\}$, $w \in Q^\infty \cup \{\varepsilon\}$ and $q, q' \in Q$ s.t. $q \neq q'$,

- $vw$ is the catenation of traces $v$ and $w$;
- $(vq) \cdot (qw) \equiv vqw$ and $(vq) \cdot (q'w) \equiv vqq'w$;
- $is(qw) \equiv q$ and $fs(vq) \equiv q$.

The $\cdot$ operator is a form of catenation which coalesces the final state of its first operand and initial state of its second operand if they are the same state. $is(v)$ is the initial state of trace $v$, and $fs(v)$ is the final state of $v$ if $v$ is finite.

**Definition:** For all $v, w \in Q^\infty$, $v \leq w$ iff $v = w$ or $v$ is a finite, initial subsequence of $w$, i.e. $w = vv'$ for some $v' \in Q^\infty$.

The relation $\leq$ is a partial order on $Q^\infty$ with the properties that $\{ v' \mid v' \leq v \}$ is totally ordered for all $v$, and every totally ordered $V \subseteq Q^\infty$ has a least upper bound in $Q^\infty$ [10], written $\lim V$.

**Notational convention:** By convention, the variable $q$ (and $q'$, etc.) shall always be understood to be an element of $Q$; the variables $r$ and $s$ (and $r'$, $s'$, etc.) shall be understood to be elements of $Q^+$; and the variables $t$ and $u$ (and $t'$, $u'$, etc.) shall be understood to be elements of $Q^\omega$. Hence a formula such as

$$\exists s, q, t(R(s, q, t))$$

will be understood to mean

$$\exists s \in Q^+, q \in Q, t \in Q^\omega(R(s, q, t))$$

## 2.2 A notation for expressing predicates on traces

In this section we present our notation for expressing properties of traces, and give its semantics.

### 2.2.1 The notation and its informal semantics

To begin with, let $V$ be some infinite set of symbols which will represent quantifiable variables, i.e. an element $v$ of $V$ will appear in expressions of the form '$\forall v\, \varphi$' and '$\exists v\, \varphi$'. Then

**Definition:** A *vocabulary* is a tuple $W = (B, U, C)$, where

1. $B$ is a set of symbols representing binary functions;
2. $U$ is a set of symbols representing unary functions;
3. $C$ is a set of symbols representing constants;
4. $B$, $U$, $C$ and $V$ are pairwise disjoint.

Such a vocabulary $W$ defines a set of *formulas*, according to the following grammar:

$$\begin{array}{lcl}
E & ::= & E_u \mid E_u\, B\, E \mid E_u\, L_b\, E \\
E_u & ::= & C \mid V \mid U\, E_u \mid L_u\, E_u \mid \text{`('}EL\text{`)'} \mid \text{`}\lceil\text{'}E\text{`}\rceil\text{'} \mid \text{`}\lfloor\text{'}E\text{`}\rfloor\text{'} \\
EL & ::= & E \mid E\text{`,'}\,EL \\
L_b & ::= & \text{`}\wedge\text{'} \mid \text{`}=\text{'} \\
L_u & ::= & \text{`}\neg\text{'} \mid \text{`}\forall\text{'}V \mid \text{`}.\text{'} \mid \text{`}\ominus\text{'} \mid \text{`}\oplus\text{'} \mid \text{`}\boxminus\text{'} \mid \text{`}\boxplus\text{'}
\end{array}$$

Where $W$ is understood we will just write $E$ for the set of formulas of $W$, and $E_u$ for the formulas produced starting with the second grammar rule above.

When writing formulas we will write '$\exists v\, \varphi$', '$\varphi_1 \vee \varphi_2$', '$\varphi_1 \Rightarrow \varphi_2$', '$\varphi_1 \Leftrightarrow \varphi_2$', and '$\varphi_1 \neq \varphi_2$' as abbreviations for the obvious expressions containing '$\forall$', '$\wedge$', '$\neg$' and '$=$', and we will feel free to use all the other common abbreviations, such as leaving out parentheses when no ambiguity results, writing '$\forall v \in S(\varphi)$' for '$\forall v(v \in S \Rightarrow \varphi)$', etc.

A formula is assigned a value for any pair $(s, t)$ s.t. $s \leq t$. If $t$ is a complete trace of a system, $s$ is the sequence of states traversed up to some moment and formula $\varphi$ is assigned the value **tt** for the pair $(s, t)$, we say that $\varphi$ holds at that moment. Then, informally,

- '$\boxplus\, \psi$' means '$\psi$ holds now and at all times in the future';
- '$\boxminus\, \psi$' means '$\psi$ holds now and at all times in the past';
- '$\lceil S \rceil$' means 'the sequence of states so far traversed is in $S$';
- '$\lfloor T \rfloor$' means 'the sequence of states which will be traversed, beginning with the present state, is in $T$';
- '$\oplus\psi$' is the value of $\psi$ in the next state, and if $\psi$ is boolean-valued it then means '$\psi$ holds in the next state';
- '$\ominus\psi$' is the value of $\psi$ in the preceding state, and if $\psi$ is boolean-valued it then means '$\psi$ held in the preceding state';

- if there is no preceding state, then $\ominus\psi$ is false.

The meaning of '.' will be explained later.

We introduce the following abbreviations:

1. '**beg**' abbreviates '$\neg\ominus\mathbf{tt}$' ('this is the initial state');
2. '$\diamondplus\varphi$' abbreviates '$\neg\boxplus\neg\varphi$' ('eventually $\varphi$ will be true');
3. '$\diamondminus\varphi$' abbreviates '$\neg\boxminus\neg\varphi$' ('$\varphi$ has been true');
4. '$\Box\varphi$' abbreviates '$\boxminus\varphi\wedge\boxplus\varphi$' ('$\varphi$ is true at all times');
5. '$\Diamond\varphi$' abbreviates '$\neg\Box\neg\varphi$' or '$\diamondminus\varphi\vee\diamondplus\varphi$' ('$\varphi$ is true at some time').

**Example:** '$\varphi$ holds in the initial state' is expressed by

$$\Box(\mathbf{beg}\Rightarrow\varphi)$$

**Example:** 'If $I$ always holds, then whenever $\varphi$ holds, eventually $\psi$ holds' is expressed by

$$\Box I\Rightarrow\Box(\varphi\Rightarrow\diamondplus\psi)$$

**Example:** 'If $\varphi$ holds infinitely often then so does $\psi$' (where by 'infinitely often' we mean that there is always a future time at which it holds) is expressed by

$$\Box\diamondplus\varphi\Rightarrow\Box\diamondplus\psi$$

**Example:** If $S\subseteq Q^+$ and $T\subseteq Q^\omega$, we define

$$S\cdot T\equiv\{\,s\cdot t\mid \mathit{fs}(s)=\mathit{is}(t)\wedge s\in S\wedge t\in T\,\}.$$

Then the formula

$$\Diamond(\lceil S\rceil\wedge\lfloor T\rfloor)$$

describes all and only the traces in $S\cdot T$.

### 2.2.2 Formal semantics of the notation

To give the formal meaning of a formula, we must first define the notion of an interpretation.

**Definition:** Given a vocabulary $W=(B,U,C)$, an *interpretation* of $W$ is a tuple $(X,Y,D,M)$, where

1. $X$ and $Y$ are nonempty sets;
2. $D\supseteq\mathsf{B}$, where $\mathsf{B}\equiv\{\mathbf{tt},\mathbf{ff}\}$;
3. Given any set $Z$, $Z^c$ is the closure of $Z$ under the operation of tupling, i.e. $Z^c$ is the smallest set $Z'\supseteq Z$ s.t. $(x_1,\ldots,x_n)\in Z'$ whenever $z_1,\ldots,z_n\in Z'$ for any $n>0$;
4. $M$ is a function which maps each $a\in C\cup V$ to some $M[\![a]\!]\in D^c$ and each $f\in B\cup U$ to some function $M[\![f]\!]\colon D^c\to D^c$.

The elements of set $X$ above are intended to represent time-varying quantities, and $Y$ the set of values they may attain, for a system whose state space (set of possible states) is $Y^X$[1]; thus, if the system's state at a certain moment is $q: X \to Y$, the value of any $x \in X$ at that moment is $q(x)$. In our notation, we write '$.\psi$' for the value of the element of $X$ denoted by $\psi$.

We will generally not fully specify the vocabularies and interpretations we use, simply assuming that the vocabularies contain any of the common mathematical symbols we may need, and that the interpretations give the appropriate meanings to these symbols. For example, we will feel free to use the symbols '$\in$', '$+$', etc., assuming that '$\in$','$+$' $\in B$, $M[\![\in]\!](x,y) = \mathbf{tt}$ if $x \in y$ and $M[\![+]\!](x,y) = x+y$. We will also abuse notation when convenient, for example identifying $a$ with the symbol $\overline{a} \in C$ s.t. $M[\![\overline{a}]\!] = a$.

Given a vocabulary $W$, interpretation $I = (X,Y,D,M)$ of $W$ and formula $\varphi$, the meaning of $\varphi$ is

$$\mathcal{P}_I[\![\varphi]\!]: Q^+ \times Q^\omega \to D^c,$$

where $Q \equiv Y^X$ and $\mathcal{P}_I$ is defined below. If $t$ is a complete trace of some system, $s \leq t$ and $\varphi \in E$, then $\mathcal{P}_I[\![\varphi]\!](s,t)$ may be thought of as the value of the formula $\varphi$ when $s$ is the sequence of states traversed so far. We now make this more precise.

**Definition:** Given vocabulary $W = (B,U,C)$ and interpretation $I = (X,Y,D,M)$ of $W$, and letting $Q \equiv Y^X$, the function $\mathcal{P}_I$ is defined as follows for all $\varphi, \varphi_i \in E$, $\psi \in E_u$, $b \in B$, $f \in U$, $v \in V \cup C$, $q \in Q$, $t \in Q^\omega$ and $s \leq t$ (here and wherever else we can do so without loss of clarity, we drop the subscript $I$):

- $\mathcal{P}[\![v]\!](s,t) \equiv M[\![v]\!]$;
- $\mathcal{P}[\![(\varphi_1, \ldots, \varphi_n)]\!](s,t) \equiv (\mathcal{P}[\![\varphi_1]\!](s,t), \ldots, \mathcal{P}[\![\varphi_n]\!](s,t))$;
- $\mathcal{P}[\![f\,\psi]\!]s \equiv M[\![f]\!](\mathcal{P}[\![\psi]\!](s,t))$;
- $\mathcal{P}[\![\psi\, b\, \varphi]\!](s,t) \equiv M[\![b]\!](\mathcal{P}[\![(\psi,\varphi)]\!](s,t))$;
- $\mathcal{P}[\![\neg\psi]\!](s,t) \equiv \mathbf{ff}$ if $\mathcal{P}[\![\psi]\!](s,t) = \mathbf{tt}$, $\mathbf{ff}$ otherwise (similarly for '$\wedge$' and '$=$');
- $\mathcal{P}[\![\forall v\, \psi]\!](s,t) \equiv \mathbf{tt}$ if $\mathcal{P}_{I'}[\![\psi]\!]s = \mathbf{tt}$ for all $d \in D^c$, where $I' \equiv (X,Y,D,M[d/v])$,[2] $\mathbf{ff}$ otherwise;
- $\mathcal{P}[\![\ominus\psi]\!](sq,t) \equiv \mathcal{P}[\![\psi]\!](s,t)$ (i.e. $\ominus\psi$ is the value of $\psi$ in the previous state);
- $\mathcal{P}[\![\ominus\psi]\!](q,t) \equiv \mathbf{ff}$;
- $\mathcal{P}[\![\oplus\psi]\!](s,t) \equiv \mathcal{P}[\![\psi]\!](sq,t)$, where $q$ is the unique element of $Q$ s.t. $sq \leq t$ (i.e. $\oplus\psi$ is the value of $\psi$ in the next state);
- $\mathcal{P}[\![.\psi]\!](s,t) \equiv \mathit{ls}(s)(\mathcal{P}[\![\psi]\!](s,t))$ if $\mathcal{P}[\![\psi]\!](s,t) \in X$ (i.e. $.\psi$ is the present value of the time-varying quantity $\psi$);
- $\mathcal{P}[\![\boxminus\psi]\!](s,t) = \mathbf{tt}$ if $\mathcal{P}[\![\psi]\!](r,t) = \mathbf{tt}$ for all $r \leq t$, $\mathbf{ff}$ otherwise (i.e. $\boxminus\psi$ is true iff $\psi$ is true now and was true at all times in the past);
- $\mathcal{P}[\![\boxplus\psi]\!](s,t) = \mathbf{tt}$ if $\mathcal{P}[\![\psi]\!](r,t) = \mathbf{tt}$ for all $r$ s.t. $s \leq r \leq t$, $\mathbf{ff}$ otherwise (i.e. $\boxplus\psi$ is true iff $\psi$ is true now and will be true at all times in the future);

[1]the set of functions from $X$ into $Y$

[2]If $g$ is a function then $g[d/v]$ indicates the function $g'$ which is the same as $g$ except that $g'(v) = d$.

- $\mathcal{P}[\![\lceil\varphi\rceil]\!](s,t) = \mathbf{tt}$ if $s \in \mathcal{P}[\![\varphi]\!](s,t) \subseteq Q^+$, $\mathbf{ff}$ otherwise (i.e. $\lceil\varphi\rceil$ holds iff $\varphi$ denotes a predicate satisfied by the sequence of states traversed so far);
- $\mathcal{P}[\![\lfloor\varphi\rfloor]\!](s,t) = \mathbf{tt}$ if
$$U \subseteq Q^\omega \wedge \exists u \in U(\mathit{fs}(s) = \mathit{is}(u) \wedge t = s \cdot u),$$
where $U \equiv \mathcal{P}[\![\varphi]\!](s,t)$, $\mathbf{ff}$ otherwise (i.e. $\lfloor\varphi\rfloor$ is true iff $\varphi$ denotes a predicate describing the future behavior of the system).

**Definition:** A *local* formula is one which contains no instance of the operators '$\lfloor\rfloor$', '$\boxplus$' or '$\oplus$'.

Note that, for any local formula $\varphi$ and any $s$, $t$ and $t'$ we have that
$$\mathcal{P}[\![\varphi]\!](s,st) = \mathcal{P}[\![\varphi]\!](s,st').$$
With this in mind, we make the following

**Definition:** For all $\varphi, \varphi' \in E$ and $t \in Q^\omega$ s.t. $\varphi'$ is a local formula,
$$\begin{array}{rcl} t \models_I \varphi & \equiv & \forall s \leq t(\mathcal{P}_I[\![\varphi]\!](s,t) = \mathbf{tt}) \\ \{\![\lfloor\varphi\rfloor]\!\}_I & \equiv & \{u \in Q^\omega \mid u \models_I \varphi\} \\ \{\![\lceil\varphi'\rceil]\!\}_I & \equiv & \{s \in Q^+ \mid \mathcal{P}_I[\![\varphi']\!](s,st') = \mathbf{tt}\} \end{array}$$
where $t'$ is some arbitrary element of $Q^\omega$. Here again we will drop the subscript $I$ whenever we can do so without loss of clarity. We write '$\models \varphi$' for '$\forall t \in Q^\omega(t \models \varphi)$'.

If $t \models \varphi$ we say that $t$ satisfies $\varphi$; if $t$ is a complete trace of the system, this means that $\varphi$ holds at all times. $\{\![\lfloor\varphi\rfloor]\!\}$ is the set of traces which satisfy $t$. If $\varphi$ is a local formula then $\{\![\lceil\varphi\rceil]\!\}$ is the set of finite traces for which $\varphi$ is true. Note that
$$\{\![\lfloor\varphi\rfloor]\!\} = \{\![\lfloor\boxplus\varphi\rfloor]\!\} = \{\![\lfloor\boxminus\varphi\rfloor]\!\} = \{\![\lfloor\square\varphi\rfloor]\!\}$$
**Example:** Let us suppose that the system in question is a digital circuit, with $X$ being the set of nodes of the circuit. In the following we implicitly use the vocabulary
$$(\{`\in'\}, \emptyset, X \cup \{`X', `\mathbf{tt}', `\mathbf{ff}'\})$$
and an interpretation $(X, \mathsf{B}, D, M)$ s.t.

1. $X \in D$;
2. $M[\![x]\!] = x$ for all $x \in X$;
3. $M[\![`X']\!] = X$;
4. $M[\![`\in']\!](x,y) = \mathbf{if}\ x \in y\ \mathbf{then\ tt\ else\ ff}$.

We express the fact that the nodes of the circuit always have boolean values by
$$\{\![\lfloor\forall x \in X(.x = \mathbf{tt} \vee .x = \mathbf{ff})\rfloor]\!\}$$
If the circuit contains a perfect inverter[3] with input $i$ and output $o$, we can express this by
$$\{\![\lfloor(\mathbf{beg} \vee \neg * o \vee .o = \neg.i) \wedge \oplus(.o = \neg.i)\rfloor]\!\}$$

[3]A perfect inverter is one which produces no 'glitch' at the output if the input changes value and changes value again before the inverter has responded to the first change.

where '$*x$' is in general an abbreviation for '$(\ominus \mathbf{tt}) \wedge .x \neq \ominus.x$' (the value of $x$ has just changed).

**Example:** The following hold for any vocabulary and interpretation, and for any $S \subseteq Q^+$, $T \subseteq Q^\omega$ and $t \in Q^\omega$:

1. $\{\lfloor\varphi_1\rfloor\} \cap \{\lfloor\varphi_2\rfloor\} = \{\lfloor\varphi_1 \wedge \varphi_2\rfloor\}$;
2. $\{\lceil\varphi_1\rceil\} \cap \{\lceil\varphi_2\rceil\} = \{\lceil\varphi_1 \wedge \varphi_2\rceil\}$;
3. $S = \{\lceil\varphi\rceil\}$ iff $\models \varphi \Leftrightarrow \lceil S \rceil$;
4. $T = \{\lfloor \mathbf{beg} \Rightarrow \lfloor T \rfloor \rfloor\}$;
5. $\models \psi \Rightarrow \diamondplus \psi \wedge \diamondminus \psi$;
6. $\models (\boxplus \psi \vee \boxminus \psi) \Rightarrow \psi$;
7. $\models \psi \Leftrightarrow \oplus \ominus \psi$;
8. $\models \mathbf{beg} \vee (\psi \Leftrightarrow \ominus \oplus \psi)$;
9. $\models \Diamond \boxplus \psi \Rightarrow \Box \diamondplus \psi$;
10. $\models (\boxplus \oplus \psi \Leftrightarrow \oplus \boxplus \psi) \wedge (\diamondplus \oplus \psi \Leftrightarrow \oplus \diamondplus \psi)$;
11. $\models (\diamondminus \ominus \psi \Leftrightarrow \ominus \diamondminus \psi)$.

Other properties may be found in [4,17] (in the latter, '$\Box$' and '$\Diamond$' are the same as our '$\boxplus$' and '$\diamondplus$').

## 2.3 Agents

Suppose that we have a program of form

$$G\ P_1 \parallel \cdots \parallel P_n,$$

where each $P_i$ describes some agent $i$, and $G$ describes some global property of the system such as the initial state. In general we define the meaning of such a program to be

$$T_G \cap T_1 \cap \cdots \cap T_n,$$

where trace set $T_G$ is the meaning of $G$, and trace set $T_i$ is the meaning of $P_i$ ($T_i$ describes the behavior of agent $i$) for $1 \leq i \leq n$. In this section we look at how to describe the behavior of an agent.

### 2.3.1 Action predicates and transition predicates

In order to describe the behavior of an agent in a manner that does not restrict the behavior of any other agent, we must be able to determine when the agent has just acted, changing some portion of the system state. To this end we introduce the notion of an action predicate:

**Definition:** An *action predicate* is any $S \subseteq Q^+$ s.t.

$$\models \lceil S \rceil \Rightarrow \exists x \in X(*x)$$

With each agent we associate an action predicate which is true when and only when the agent has just acted; we refer to this as *the* action predicate of the agent.

We need a way of describing what the results of a particular action by an agent will be. For this purpose we can use some $S \subseteq Q^+$ with this interpretation: if the action takes place at a moment when $s$ is the sequence of states traversed so far, then there will be a transition to some state $q$ s.t. $sq \in S$. A trace set $S \subseteq Q^+$ used in this way we call a ***transition predicate***. We say that the agent ***performs*** $S$ if it performs such an action.

As an example, suppose we have an agent which may only change the values of elements of $L \subseteq X$, and that no other agent may change these values. Then the agent's action predicate is

$$\{\lceil \exists x \in L(*x) \rceil\}.$$

Furthermore, to describe the action 'invert the (boolean) value of b' by this agent, we use the transition predicate

$$\{\lceil .b = \neg \ominus .b \wedge \forall x \in L(*x \Rightarrow x = b) \rceil\}.$$

Note that this allows the value of any $x \in X - L$ to change; this is to allow for the possibility (however unlikely) that one or more other agents act at precisely the same time as this one.

### 2.3.2 A simple notation for describing agents

We use what may be considered a generalization of the programming notation used in [6], to describe the behavior of some simple agents. Informally, an expression of the form

$$SA: \ [e_1 \xrightarrow{\mathrm{f}} c_1 \mid \cdots \mid e_n \xrightarrow{\mathrm{f}} c_n \mid e_{n+1} \xrightarrow{\mathrm{j}} c_{n+1} \mid \cdots \mid e_m \xrightarrow{\mathrm{j}} c_m \mid \alpha : a \mid \varepsilon : e]$$

(where $a$, $e$, each $e_i$ and each $c_i$ are local formulas) defines an agent with action predicate $\{\lceil a \rceil\}$ which, as long as the error condition $e$ remains false, is for any $i$ enabled to perform $\{\lceil c_i \rceil\}$ whenever the corresponding $e_i$ holds. Furthermore, if $e_i$ $(1 \leq i \leq n)$ holds infinitely often then the agent infinitely often performs $\{\lceil c_i \rceil\}$, and the agent performs $\{\lceil c_j \rceil\}$ $(n < j \leq m)$ whenever $e_j$ holds and continues to hold 'long enough' (i.e. it is forbidden that $e_j$ begin to hold and continue to hold forever without the agent performing $\{\lceil c_j \rceil\}$). Using the terminology of [14], $\{\lceil c_i \rceil\}$ is chosen to be performed in a fair (strongly fair) manner or just (weakly fair) manner, according as $i \leq n$ or $i > n$.

We consider $SA$ to be an abbreviation for

$$(a \wedge \ominus \boxminus \neg e \Rightarrow \bigvee_{i=1}^{m} (c_i \wedge \ominus e_i)) \wedge (\Box \neg e \Rightarrow \bigwedge_{i=1}^{n} F_i \wedge \bigwedge_{i=n+1}^{m} J_i)$$

where for all $i$,

$$\begin{aligned} F_i &\equiv \Box \diamondplus e_i \Rightarrow \Box \diamondplus (e_i \wedge \oplus c_i) \\ J_i &\equiv \Box \diamondplus (\neg e_i \vee (e_i \wedge \oplus c_i)) \\ &\Leftrightarrow \Box \diamondplus (\neg e_i \vee c_i) \end{aligned}$$

As an example, a non-perfect inverter with input $i$ and output $o$ may be described by

$$\{\lfloor [.i = .o \xrightarrow{\mathrm{j}} *o \mid \alpha : *o \mid \varepsilon : *i \wedge \ominus(.i = .o)] \rfloor\}$$

## 2.4 Semantics of two simple concurrent languages

### 2.4.1 Digital circuits

Our first concurrent language is a simple notation for describing asynchronous digital circuits composed of five types of primitive elements. Let $N$ be a set of symbols used to identify nodes in the circuit. The syntax of the notation is then

$$PR ::= PR \parallel PR \mid \text{`}I\text{'}(N;N) \mid \text{`}A\text{'}(N,N;N) \mid \text{`}O\text{'}(N,N;N) \mid \text{`}C\text{'}(N,N;N) \mid \text{`}R\text{'}(N,N;N,N)$$

'$I(a;b)$' indicates an inverter with input $a$ and output $b$; '$A(a,b;c)$' indicates an AND gate with inputs $a$ and $b$ and output $c$, and similarly for '$O(a,b;c)$'. '$C(a,b;c)$' indicates a C-element[4] with inputs $a$ and $b$ and output $c$. '$R(r_1,r_2;a_1,a_2)$' indicates a fair arbiter[5] with request-acknowledge pairs $r_1, a_1$ and $r_2, a_2$. We add the restriction that no $n \in N$ may appear twice as an output in a program.

We will write '$\xi(e_1,e_2)$' for '$\neg e_1 \wedge \ominus(e_1 \wedge \neg e_2)$'. $\xi(e_1,e_2)$ indicates a condition which may produce a 'glitch' or 'runt pulse' in the output of some logic element; for example, for an AND gate with inputs $a$ and $b$ and output $c$, if $\xi(.a \wedge .b, .c)$ ever holds this says that the gate has been excited to change its output to high, but the excitation was removed before it could do so.

We define

$$elem[\![e_1 \mid e_2 \mid n]\!] \equiv \{\![\varphi]\!\}$$

where

$$\begin{aligned}\varphi \;\equiv\; & [\, e_1 \wedge \neg.n \xrightarrow{\mathsf{j}} .n \\ & \mid e_2 \wedge .n \xrightarrow{\mathsf{j}} \neg.n \\ & \mid \alpha : *n \mid \varepsilon : \xi(e_1, .n) \vee \xi(e_2, \neg.n) \\ & ].\end{aligned}$$

$elem[\![e_1 \mid e_2 \mid n]\!]$ describes a logic element with an output $n$ which becomes true when $e_1$ holds, false when $e_2$ holds, and otherwise does not change.

The meaning of a program $P \in PR$ is then given by the semantic function $\mathcal{M}: PR \to \wp(Q^\omega)$,[6] where $Q \equiv \mathsf{B}^N$ and

- $\mathcal{M}[\![P_1 \parallel P_2]\!] \equiv \mathcal{M}[\![P_1]\!] \cap \mathcal{M}[\![P_2]\!]$;
- $\mathcal{M}[\![I(a;b)]\!] \equiv elem[\![\neg.a \mid .a \mid b]\!]$;
- $\mathcal{M}[\![A(a,b;c)]\!] \equiv elem[\![.a \wedge .b \mid \neg(.a \wedge .b) \mid c]\!]$;
- $\mathcal{M}[\![O(a,b;c)]\!] \equiv elem[\![.a \vee .b \mid \neg(.a \vee .b) \mid c]\!]$;
- $\mathcal{M}[\![C(a,b;c)]\!] \equiv elem[\![.a \wedge .b \mid \neg.a \wedge \neg.b \mid c]\!]$;

[4]The output of a $C$ element goes high when both its inputs are high, low when both its inputs are low, and otherwise remains the same.

[5]An arbiter grants mutually exclusive access to some resource via a set of request-acknowledge pairs. When a device wishes to use the resource, it raises its request, and once it has obtained the resource it may release it by lowering the request. The arbiter grants a device access to the resource by raising the corresponding acknowledge, which remains high until the request goes low and the resource is released.

[6]We write $\wp(Y)$ for the power set of a set $Y$

- $\mathcal{M}[\![R(r_1, r_2; a_1, a_2)]\!] \equiv \{\![\varphi]\!\}$, where

$$\begin{aligned}
\varphi \;\equiv\;& [\ .r_1 \wedge \neg.a_1 \wedge \neg.a_2 \xrightarrow{\mathsf{f}} .a_1 \wedge \neg.a_2 \\
& \mid .r_2 \wedge \neg.a_1 \wedge \neg.a_2 \xrightarrow{\mathsf{f}} .a_2 \wedge \neg.a_1 \\
& \mid \neg.r_1 \wedge .a_1 \xrightarrow{\mathsf{j}} \neg.a_1 \wedge \neg * a_2 \\
& \mid \neg.r_2 \wedge .a_2 \xrightarrow{\mathsf{j}} \neg.a_2 \wedge \neg * a_1 \\
& \mid \alpha : *a_1 \vee *a_2 \mid \varepsilon : e \\
& ] \\
e \;\equiv\;& \xi(.r_1, .a_1) \vee \xi(.r_2, .a_2) \vee \xi(\neg.r_1, \neg.a_1) \vee \xi(\neg.r_2, \neg.a_2)
\end{aligned}$$

### 2.4.2 Petri nets

Our second concurrent language is that of Petri nets [21]. A Petri net is a graph whose nodes are of two types—*places* and *transitions*—such that every arc connects two nodes of different types. For a place $p$ and transition $t$, if there is an arc from $p$ to $t$ (resp. $t$ to $p$) then we say that $p$ is an *input place* (resp. *output place*) of $t$. The set of output places of transition $t$ is denoted $t^\bullet$, and its set of input places is denoted $^\bullet t$. We imagine that each place may hold a number of tokens. The distribution of tokens is modified when a transition *fires*, removing one token from each input place and adding a token to each output place. A transition may fire only when it is enabled, i.e. when all of its input places contain tokens. Note that a transition may, by firing, disable another transition from firing. Two transitions $t$ and $t'$ cannot fire at the same time if there is a place $p$ that they both affect, i.e. if $p \in t^\bullet \cup {}^\bullet t$ and $p \in t'^\bullet \cup {}^\bullet t'$; we write $t \bowtie t'$ if this is so. The choice of transition firings is strongly fair.

Formally, a (marked) Petri net is a tuple $(P, T, R, M)$, where $P$ is the set of places, $T$ is the set of transitions, $R \subseteq P \times T \cup T \times P$ is the set of arcs, $M\colon P \to \mathsf{N}$[7] gives the number of tokens initially in each place, and $P \cap T = \emptyset$. We will furthermore assume that $P$ and $T$ are finite.

We will use the state space $Q \equiv \mathsf{N}^{P \cup T}$; for any $q \in Q$, $p \in P$ and $t \in T$, $q(p)$ is the number of tokens in place $p$ and $q(t)$ is the number of times transition $t$ has fired. We will write '$x \uparrow$' for '$.x = \ominus.x + 1$', '$x \downarrow$' for '$.x = \ominus.x - 1$'.

The behavior of a Petri net $(P, T, R, M)$ is then

$$\{\![(\mathbf{beg} \Rightarrow \varphi) \wedge ME \wedge \bigwedge_{p \in P} \theta_p \wedge \bigwedge_{t \in T} \psi_t]\!\},$$

where

1. $\varphi \equiv \bigwedge_{p \in P}(.p = M(p)) \wedge \bigwedge_{t \in T}(.t = 0)$;
2. $ME \equiv \bigwedge_{t \bowtie t'} \neg(t \uparrow \wedge t' \uparrow)$;
3. $\theta_p \equiv *p \Rightarrow \bigvee_{t \in T}(*t \wedge p \in O_t \cup J_t)$;
4. $\psi_t \equiv [\ \bigwedge_{i \in I_t} .i > 0 \xrightarrow{\mathsf{f}} t \uparrow \wedge \bigwedge_{p \in O_t} p \uparrow \wedge \bigwedge_{p \in J_t} p \downarrow \mid \alpha : *t \mid \varepsilon : \mathbf{ff}\ ]$;
5. $I_t \equiv {}^\bullet t$, $O_t \equiv t^\bullet - {}^\bullet t$ and $J_t \equiv {}^\bullet t - t^\bullet$.

[7] $\mathsf{N}$ is the set of natural numbers

# Chapter 3

# Satisfiability

## 3.1 Initial conditions and evolution conditions

We now address the question of how to guarantee that a concurrent program denotes a *nonempty* trace set, or, equivalently, a satisfiable predicate on traces. As we have seen with the simple programs of Chapter 2, the meaning $T$ of a program will in general be of the form

$$T \equiv T_G \cap \bigcap_{i \in I} T_i,$$

where $I$ is the set of agents, $T_i$ is the meaning of the subprogram describing agent $i$ for all $i \in I$, and $T_G$ is some general constraint on the system, not corresponding to any particular agent. We will therefore need to show that certain kinds of set intersections are nonempty.

As a separation of concerns, we shall find it convenient to express $T_G$ as the intersection of two trace sets, one restricting only the initial state (called an initial condition), and the other not restricting the initial state at all (called an evolution condition). Formally,

**Definition:** An *initial condition* is any trace set of form

$$\{\lfloor \mathbf{beg} \Rightarrow \lceil S \rceil \rfloor\}$$

(which we will write as $S^{\text{in}}$) for some $S \subseteq Q^+$. Choosing $S \equiv \{\lceil \varphi \rceil\}$, we see that $\{\lfloor \mathbf{beg} \Rightarrow \varphi \rfloor\}$ is an initial condition for any local formula $\varphi$. An *evolution condition* is any $T \subseteq Q^\omega$ s.t.

$$\forall q \in Q \, \exists t \in T(is(t) = q).$$

Note that evolution conditions are always nonempty, since $Q \neq \emptyset$, and that any superset of an evolution condition is also an evolution condition.

**Property 1** *Any $T \subseteq Q^\omega$ can be expressed as the intersection of an initial condition and an evolution condition.*

**Proof:** Let

$$\begin{aligned} S &\equiv \{\, q \mid \exists t \in T(q = is(t)) \,\} \\ T' &= T \cup (Q^\omega - S^{\text{in}}) \end{aligned}$$

$T'$ is an evolution condition, since for all $q$,

$$\begin{aligned} \neg\exists t \in T(q = is(t)) &\Leftrightarrow q \notin S \\ &\Rightarrow \forall u \in Q^\omega(qu \notin S^{\text{in}}) \\ &\Rightarrow \forall u \in Q^\omega(qu \in T') \\ &\Rightarrow \exists u \in Q^\omega(qu \in T') \end{aligned}$$

Then since $T \subseteq S^{\text{in}}$ we have that $T = S^{\text{in}} \cap T'$. ∎

Let $T_G \equiv S_G^{\text{in}} \cap T'_G$, where $S_G$ describes the initial state of the system; then

$$T = S_G^{\text{in}} \cap (T'_G \cap \bigcap_{i \in I} T_i).$$

Given any evolution condition $T'$ and any $S \subseteq Q^+$ s.t. $S \cap Q$ is nonempty, we have that $T' \cap S^{\text{in}}$ is nonempty, since for any $q \in S$ there is some $t \in T'$ with $is(t) = q$, and this implies that $t \in S^{\text{in}}$. $T$ is then nonempty if

1. $T'_G \cap \bigcap_{i \in I} T_i$ is an evolution condition;

2. $Q \cap S_G$ is nonempty, i.e. there is some state which satisfies $S_G$.

For the language of digital circuits presented in Chapter 2, we had $S_G = \{\![\mathbf{tt}]\!\}$ for all agents $i$, thus trivially satisfying condition 2. above. For the Petri net semantics presented in the same chapter, we had

$$S_G = \{\![\bigwedge_{p \in P} (.p = M(p)) \wedge \bigwedge_{t \in T} (.t = 0)]\!\};$$

in this case it is also easily seen that condition 2. is satisfied. Condition 2. is easily checked in all the cases we consider, and so henceforth we focus on requirement 1.

## 3.2 Cores

To prove that requirement 1 is satisfied we will need to use the notion of an *A-core* of a trace set, where $A$ is an action predicate; in this section we develop this notion.

In Chapter 2 we described the behavior of an agent by an expression of the form

$$SA: \ [e_1 \xrightarrow{\mathsf{f}} c_1 \mid \cdots \mid e_n \xrightarrow{\mathsf{f}} c_n \mid e_{n+1} \xrightarrow{\mathsf{j}} c_{n+1} \mid \cdots \mid e_m \xrightarrow{\mathsf{j}} c_m \mid \alpha : a \mid \varepsilon : e]$$

Each of the pairs $(e_i, c_i)$ is a rule allowing the agent to perform $\{\![c_i]\!\}$ whenever $e_i$ holds. Each pair $(e_i, c_i)$ corresponds to a formal object which we call a transition rule:

**Definition:** A *transition rule* is a pair $l = (\pi, \tau)$ s.t. $\pi, \tau \subseteq Q^+$. Given some action predicate $A \subseteq Q^+$, we say that $l$ is *A-admissible* iff

1. $\forall s \in \pi \, \exists q(sq \in \tau)$, and

2. $\models \lceil \tau \rceil \wedge \ominus \lceil \pi \rceil \Rightarrow \lceil A \rceil \vee \mathbf{null}$,

where

$$\mathbf{null} \equiv \neg\exists x \subset X(*x) \wedge \ominus\mathbf{tt}$$

(i.e. $\{\lceil \mathbf{null} \rceil\}$ is the set of $sq$ s.t. $q = \mathit{fs}(s)$). Note that a transition rule is $A$-admissible if it is $A'$-admissible for some $A' \subseteq A$.

In the expression $SA$ above, we would expect that whenever some $e_i$ holds, any state transition allowed by $c_i$ should be either a null transition or should result in $a$ holding, since $\{\lceil a \rceil\}$ is the action predicate of the agent being described. This is equivalent to saying that $(\{\lceil e_i \rceil\}, \{\lceil c_i \rceil\})$ should be $\{\lceil a \rceil\}$-admissible.

A trace set defined by an expression of form $SA$ is a special case of an $\{\lceil a \rceil\}$-founded trace set:

**Definition:** If $A \subseteq Q^+$ is an action predicate, we say that a trace set $T'$ is *A-founded* iff there exists a countable set $L$ of $A$-admissible transition rules s.t. $T' \supseteq G(L, A)$, where

$$\begin{aligned} G(L,A) &\equiv \mathit{safe}(L,A) \cap \bigcap_{l \in L} \mathit{fair}(l) \\ \mathit{safe}(L,A) &\equiv \{\lfloor \lceil A \rceil \Rightarrow \exists(\pi,\tau) \in L(\lceil \tau \rceil \wedge \ominus\lceil \pi \rceil) \rfloor\} \\ \mathit{fair}(\pi,\tau) &\equiv \{\lfloor \Box\diamondsuit\lceil \pi \rceil \Rightarrow \Box\diamondsuit(\lceil \pi \rceil \wedge \oplus\lceil \tau \rceil) \rfloor\} \end{aligned}$$

We say that $L$ is an *A-core* of $T'$.

Examining the definitions, it is easily seen that $\{\lfloor SA \rfloor\}$ is $\{\lceil a \rceil\}$-founded if for all $i$,

1. $(\{\lceil e_i \rceil\}, \{\lceil c_i \rceil\})$ is $\{\lceil a \rceil\}$-admissible, and

2. $\mathit{fair}(l) \subseteq \mathit{just}(l)$, where $l \equiv (\{\lceil e_i \rceil\}, \{\lceil c_i \rceil\})$ and for any transition rule $(\pi, \tau)$,

$$\mathit{just}(\pi,\tau) \equiv \{\lfloor \Box\diamondsuit(\neg\lceil \pi \rceil \vee \lceil \tau \rceil) \rfloor\}.$$

Condition 1. above is sufficient, due to the following.

**Property 2** *For any transition rule* $(\pi, \tau)$,

$$\mathit{fair}(\pi,\tau) \subseteq \mathit{just}(\pi,\tau).$$

**Proof:** For any trace $t$,

$$\begin{aligned} t \models \neg\Box\diamondsuit\lceil \pi \rceil \quad &\Leftrightarrow \quad t \models \Diamond\boxplus\neg\lceil \pi \rceil \\ &\Rightarrow \quad t \models \Box\diamondsuit\neg\lceil \pi \rceil \\ &\Rightarrow \quad t \models \Box\diamondsuit(\neg\lceil \pi \rceil \vee \lceil \tau \rceil) \\ t \models \Box\diamondsuit(\lceil \pi \rceil \wedge \oplus\lceil \tau \rceil) \quad &\Rightarrow \quad t \models \Box\diamondsuit\oplus\lceil \tau \rceil \\ &\Rightarrow \quad t \models \Box\diamondsuit\lceil \tau \rceil \\ &\Rightarrow \quad t \models \Box\diamondsuit(\neg\lceil \pi \rceil \vee \lceil \tau \rceil) \end{aligned}$$

Since $t \in \mathit{fair}(\pi,\tau)$ iff

$$(t \models \neg\Box\diamondsuit\lceil \pi \rceil) \vee (t \models \Box\diamondsuit(\lceil \pi \rceil \wedge \oplus\lceil \tau \rceil))$$

we then have that $t \in \mathit{fair}(\pi,\tau) \Rightarrow t \in \mathit{just}(\pi,\tau)$. ∎

Note that in the definition of an $A$-core $L$ we allow $L$ to be an *infinite* set of transition rules. We will find this to be very convenient later on when we deal with recursion.

## 3.3 The main theorem

Returning to the discussion of section 3.1, we want to show that

$$T'_G \cap \bigcap_{i \in I} T_i$$

is an evolution condition. If $T'_G$ is weaker than some constraint on what actions are possible, i.e.

$$T'_G \supseteq \{\lfloor \mathbf{beg} \vee \mathbf{null} \vee \lceil A \rceil \rfloor\}$$

for some action predicate $A$, then the following theorem shows that it is sufficient that $\bigcap_i T_i$ be $A$-founded.

**Theorem 3** $\Sigma(A) \cap T'$ *is an evolution condition for any action predicate $A$ and $A$-founded trace set $T'$, where*

$$\Sigma(A) \equiv \{\lfloor \mathbf{beg} \vee \mathbf{null} \vee \lceil A \rceil \rfloor\}.$$

**Proof:** We show that $\Sigma(A) \cap T'$ is an evolution condition by constructing, for any $q \in Q$, some $t \in \Sigma(A) \cap T'$ s.t. $q = is(t)$, as follows:

First, we make the following definitions:

1. $L$ is some $A$-core of $T'$.
2. $f\colon \mathsf{N} \to L$ is a function s.t. $\{\, n \in \mathsf{N} \mid f(n) = l \,\}$ is infinite for any $l \in L$. It is a straightforward exercise to show that such a function exists. We shall use $f$ as a means of 'scheduling' the transition rules in $L$.
3. $S \equiv \{\lceil (\mathbf{beg} \vee \mathbf{null} \vee \lceil A \rceil) \wedge (\lceil A \rceil \Rightarrow \exists (\pi, \tau) \in L(\lceil \tau \rceil \wedge \ominus \lceil \pi \rceil)) \rceil\}$.
   It is easily seen that for any $t \in Q^\omega$,
   $$t \in \Sigma(A) \cap \mathit{safe}(L, A) \Leftrightarrow \forall s \leq t (s \in S).$$
4. $S' \equiv \{\lceil \boxminus \lceil S \rceil \rceil\}$.
5. For any $l = (\pi, \tau) \in L$ and $s \in \pi$, $\chi(s, l)$ is some $sq' \in \tau$ (such a $q'$ exists since $l$ is $A$-admissible).
6. For all $i \in \mathsf{N}$,
   (a) $s_0 \equiv q$;
   (b) $s_{i+1} \equiv \chi(s', f(i))$ for some $s' \geq s_i$ s.t. $s' \in \pi \cap S'$, where $f(i) = (\pi, \tau)$, if such an $s'$ exists;
   (c) $s_{i+1} = s_i q'$ otherwise, where $q' \equiv \mathit{fs}(s_i)$.

Let $t \equiv \lim\{\, s_i \mid i \in \mathsf{N} \,\}$. Obviously, $is(t) = q$. To show that $t \in T'$ it suffices to show that $t \in \Sigma(A) \cap \mathit{safe}(L, A)$ and $t \in \mathit{fair}(l)$ for all $l \in L$. We have that

1. $q \in S$,
2. $\chi(s, l) \in S$ whenever $s \in \pi$ for any $l = (\pi, \tau) \in L$, and

3. $sq' \in S$ if $q' = fs(s)$;

by induction we then have that $\forall i \forall s \leq s_i(s \in S)$, hence $s \in S$ for any $s \leq t$[1], and so $t \in \Sigma(A) \cap \mathsf{safe}(L, A)$.

It remains only to show that $t \in \mathit{fair}(l)$ for all $l \in L$. Given any $(\pi, \tau) \in L$, suppose that

$$F: \qquad \forall s \leq t \exists s'(s \leq s' \leq t \wedge s' \in \pi)$$

(i.e. $t \models \Box \diamondplus \lceil \pi \rceil$). Then, since $r \in S$ for all $r \leq t$,

$$\forall s \leq t \exists s'(s \leq s' \leq t \wedge s' \in \pi \cap S'),$$

and so

$$\forall s \leq t \exists s' \geq s(s' \in \pi \cap S').$$

Hence for all $i$ there is some $s' \geq s_i$ s.t. $s' \in \pi \cap S'$. Then, noting that

1. for all $i$ there is some $j \geq i$ s.t. $f(j) = (\pi, \tau)$, and hence there exist $s'$ and $q$ s.t. $s_i \leq s'$, $s'q \leq t$, $s' \in \pi$ and $s'q \in \tau$;

2. for all $s \leq t$ there is some $i$ s.t. $s \leq s_i$;

we have that

$$G: \qquad \forall s \leq t \exists s', q(s \leq s' \wedge s' \in \pi \wedge s'q \leq t \wedge s'q \in \tau)$$

(i.e. $t \models \Box \diamondplus (\lceil \pi \rceil \wedge \oplus \lceil \tau \rceil)$). So $F \Rightarrow G$, which is the same as

$$t \models \Box \diamondplus \lceil \pi \rceil \Rightarrow \Box \diamondplus (\lceil \pi \rceil \wedge \oplus \lceil \tau \rceil),$$

i.e. $t \in \mathit{fair}(\pi, \tau)$. ∎

The above theorem forms the core of our approach to ensuring the satisfiability of the predicate denoted by a program.

## 3.4 The intersection theorem

For each agent $i \in I$, let us choose an action set $A_i$ s.t. $T_i$ is $A_i$-founded and $A_i \cap A_j = \emptyset$ for all $j \neq i$. If $A_i \subseteq A$ for all $i$, then the following theorem shows that $\bigcap_i T_i$ is $A$-founded.

This theorem may not seem to be applicable to the two programming languages discussed in Chapter 2, since the action sets of distinct agents are definitely *not* disjoint. However, we shall see shortly how to get around this.

We now give the theorem.

**Theorem 4** *Let $I$ be a countable set and for all $i \in I$ let $T_i \subseteq Q^\omega$ be $A_i$-founded, where $A_i$ is an action predicate. If $A_i \cap A_j = \emptyset$ for all $i \neq j$, then $\bigcap_i T_i$ is $(\bigcup_i A_i)$-founded.*

**Proof:** For all $i \in I$ let $L_i$ be an $A_i$-core of $T_i$; furthermore, let $A \equiv \bigcup_i A_i$, $L \equiv \bigcup_i L_i$ and $T' \equiv \bigcap_i T_i$.

[1]using the fact that, for any totally ordered $V \subseteq Q^\infty$, $s \leq \lim V \Leftrightarrow \exists v \in V(s \leq v)$.

Since each $L_i$ is a countable set of $A_i$-admissible transition rules, $L$ is a countable set of $A$-admissible transition rules. In addition, using the fact that $A_i \cap A_j = \emptyset$ for all $i \neq j$, we have for any $t \in \mathit{safe}(L, A)$, $i \in I$ and $s \leq t$ that

$$\begin{array}{rcl} s \in A_i & \Rightarrow & s \in \{[\lceil A_i \rceil \wedge (\lceil A \rceil \Rightarrow \exists j \exists (\pi, \tau) \in L_j(\lceil \tau \rceil \wedge \ominus \lceil \pi \rceil))]\} \\ & \Rightarrow & s \in \{[\exists j \exists (\pi, \tau) \in L_j(\lceil A_i \rceil \wedge \lceil \tau \rceil \wedge \ominus \lceil \pi \rceil)]\} \\ & \Rightarrow & s \in \{[\exists j \exists (\pi, \tau) \in L_j(\lceil A_i \rceil \wedge \lceil A_j \rceil \wedge \lceil \tau \rceil \wedge \ominus \lceil \pi \rceil)]\} \\ & \Rightarrow & s \in \{[\exists j \exists (\pi, \tau) \in L_j(i = j \wedge \lceil \tau \rceil \wedge \ominus \lceil \pi \rceil)]\} \\ & \Rightarrow & s \in \{[\exists (\pi, \tau) \in L_i(\lceil \tau \rceil \wedge \ominus \lceil \pi \rceil)]\} \end{array}$$

and hence $t \in \mathit{safe}(L_i, A_i)$. So $\mathit{safe}(L, A) \subseteq \mathit{safe}(L_i, A_i)$ for all $i \in I$. Then

$$\begin{array}{rcl} T' & \supseteq & \bigcap_i (\mathit{safe}(L_i, A_i) \cap \bigcap_{l \in L_i} \mathit{fair}(l)) \\ & = & \bigcap_i \mathit{safe}(L_i, A_i) \cap \bigcap_{l \in L} \mathit{fair}(l) \\ & \supseteq & \mathit{safe}(L, A) \cap \bigcap_{l \in L} \mathit{fair}(l) \end{array}$$

i.e. $L$ is an $A$-core of $T'$, and so $T'$ is $A$-founded. ∎

## 3.5 Non-disjoint action predicates

We now show how to prove satisfiability when the agents of the system being described have non-disjoint action predicates. We begin with one more definition.

**Definition:** A trace set $T'$ is $(A, A')$-founded ($A$ and $A'$ being action predicates) iff there exists an $A$-core $L$ of $T'$ s.t. each element of $L$ is $A'$-admissible. Such an $L$ is called an $(A, A')$-core of $T'$.

Generally, $A$ will be the agent's action predicate, and $A'$ will be a subset of $A$ indicating a condition when *only* this agent, and no other, has just acted.

The following property is useful in showing that a trace set is $(A, A')$-founded.

**Property 5** *Trace set $T'$ is $(A, A')$-founded if there exists an $A$-core $L$ of $T'$ s.t.*

$$\forall (\pi, \tau) \in L \, \exists \tau' \subseteq Q^+((\models \ominus \lceil \pi \rceil \wedge \lceil \tau' \rceil \Rightarrow \lceil \tau \rceil) \wedge (\pi, \tau') \textit{ is } A'\textit{-admissible}).$$

**Proof:** Let $L = \{ (\pi_i, \tau_i) \mid i \in \mathsf{N} \}$, and for all $i$ define

$$\tau_i' \equiv \text{some transition predicate s.t. } \models \ominus \lceil \pi \rceil \wedge \lceil \tau' \rceil \Rightarrow \lceil \tau \rceil \text{ and } (\pi_i, \tau_i') \text{ is } A'\text{-admissible.}$$

Letting $L' \equiv \{ (\pi_i, \tau_i') \mid i \in \mathsf{N} \}$, we have that

$$G(L', A) \subseteq G(L, A) \subseteq T'$$

since $\tau_i' \subseteq \tau_i$ for all $i$. So $L'$ is an $A$-core of $T'$, and by the definition of $\tau_i'$ it is also an $(A, A')$-core of $T'$. ∎

**Theorem 6** *Let $I$ be a countable set, $T'_G$ a trace set, and for all $i \in I$ let $A_i, A'_i$ be action predicates and $T_i$ a trace set s.t.*

1. $A'_i \cap A'_j = \emptyset$ *for all* $j \neq i$, $j \in I$;
2. $\models (\mathbf{beg} \vee \mathbf{null} \vee \exists j \in I \lceil A'_j \rceil) \wedge \lceil A_i \rceil \Rightarrow \lceil A'_i \rceil$;
3. $\{\lfloor \mathbf{beg} \vee \mathbf{null} \vee \exists j \in I \lceil A'_j \rceil \rfloor\} \subseteq T'_G$;
4. $T_i$ *is* $(A_i, A'_i)$*-founded.*

*Then* $T'_G \cap \bigcap_{i \in I} T_i$ *is an evolution condition.*

**Proof:** For all $i$, let $L_i$ be an $(A_i, A'_i)$-core of $T_i$. Then, using 2. above,

$$\begin{aligned}
\Sigma(\bigcup_j A'_j) \cap \mathit{safe}(L_i, A'_i) &= \{\lfloor (\mathbf{beg} \vee \mathbf{null} \vee \exists j \lceil A'_j \rceil) \wedge (\lceil A'_i \rceil \Rightarrow \exists (\pi, \tau) \in L_i(\lceil \tau \rceil \wedge \ominus \lceil \pi \rceil)) \rfloor\} \\
&\subseteq \{\lfloor (\mathbf{beg} \vee \mathbf{null} \vee \exists j \lceil A'_j \rceil) \wedge (\lceil A_i \rceil \Rightarrow \exists (\pi, \tau) \in L_i(\lceil \tau \rceil \wedge \ominus \lceil \pi \rceil)) \rfloor\} \\
&= \Sigma(\bigcup_j A'_j) \cap \mathit{safe}(L_i, A_i).
\end{aligned}$$

Hence, using this and 3. above,

$$\begin{aligned}
\Sigma(\bigcup_j A'_j) \cap G(L_i, A'_i) &\subseteq \Sigma(\bigcup_j A'_j) \cap G(L_i, A_i) \\
&\subseteq T'_G \cap T_i.
\end{aligned}$$

Then

$$\Sigma(\bigcup_i A'_i) \cap \bigcap_i G(L_i, A'_i) \subseteq T'_G \cap \bigcap_i T_i$$

and since Theorem 3 tells us that the left-hand side of the above inclusion is an evolution condition, so is the right-hand side. ∎

In summary then, we show that

$$S_G^{\mathrm{in}} \cap T'_G \cap \bigcap_{i \in I} T_i$$

is nonempty by finding action predicates $A_i$ and $A'_i$ for all $i$ s.t.

1. there is some state satisfying $S_G$;
2. $\{\lfloor (\mathbf{beg} \Rightarrow \lceil S_G \rceil) \wedge (\mathbf{beg} \vee \mathbf{null} \vee \exists j \lceil A'_j \rceil) \rfloor\} \subseteq T_G$;
3. $A'_i \cap A'_j = \emptyset$ for all $j \neq i$, $i, j \in I$;
4. $\models (\mathbf{beg} \vee \mathbf{null} \vee \exists j \in I \lceil A'_j \rceil) \wedge \lceil A_i \rceil \Rightarrow \lceil A'_i \rceil$ for all $i$;
5. $T_i$ is $(A_i, A'_i)$-founded, for all $i$.

## 3.6 Two proofs of satisfiability

Let

$$\langle x_1 := e_1, \ldots, x_n := e_n \rangle$$

abbreviate

$$\ominus \mathbf{tt} \wedge \forall x \in X(\bigwedge_{i=1}^{n}(x \neq \ominus x_i \wedge \neg * x) \vee \bigvee_{i=1}^{n}(x = \ominus x_i \wedge .x = \ominus e_i)),$$

i.e. there is a previous state, and it is the same as the present state except that the values of the various $x_i$ have been changed in the manner indicated. Furthermore, let '$x := e$' abbreviate '$\langle x := e \rangle$'.

We first look at the digital circuit language of the previous chapter. This case is simple enough that it doesn't require all the steps listed in the previous section.

For all $n \in N$, let

$$a_n \equiv n := \neg .n,$$

i.e. $\{\!\lceil a_n \rceil\!\}$ is an action predicate indicating that the value of node $n$ has just changed, while the values of all other nodes of the circuit remained unchanged.

Using Property 5 and that fact that, for any $n \in N$,

$$\models \ominus \neg .n \wedge n := \neg .n \Rightarrow .n \wedge a_n$$
$$\models \ominus .n \wedge n := \neg .n \Rightarrow \neg .n \wedge a_n,$$

it is easily seen from their definitions that

1. $\mathcal{M}[\![A(x,y;n)]\!]$, $\mathcal{M}[\![O(x,y;n)]\!]$, $\mathcal{M}[\![C(x,y;n)]\!]$ and $\mathcal{M}[\![I(x;n)]\!]$ are all $(\{\!\lceil *n \rceil\!\}, \{\!\lceil a_n \rceil\!\})$-founded,
2. $\mathcal{M}[\![R(x,y;n_1,n_2)]\!]$ is $(\{\!\lceil *n_1 \vee *n_2 \rceil\!\}, \{\!\lceil a_{n_1} \vee a_{n_2} \rceil\!\})$-founded,
3. $\{\!\lceil a_n \rceil\!\} \cap \{\!\lceil a_m \rceil\!\} = \emptyset$ for all $n \neq m$, $n, m \in N$, and
4. $\models (\mathbf{beg} \vee \mathbf{null} \vee \exists m(a_m)) \wedge *n \Rightarrow a_n$ for all $n \in N$.

Given a program $P_1 \parallel \cdots \parallel P_k$, let $N_i$ be the set of outputs of the component $P_i$ for all $i$. Note that $N_i$ has either one or two elements, and that $N_i \cap N_j = \emptyset$ for all $i \neq j$. Defining

1. $A_i \equiv \{\!\lceil \bigvee_{n \in N_i} *n \rceil\!\}$ and
2. $A_i' \equiv \{\!\lceil \bigvee_{n \in N_i} a_n \rceil\!\}$,

we see from 1–4 above that

1. $\mathcal{M}[\![P_i]\!]$ is $(A_i, A_i')$-founded for all $i$,
2. $A_i' \cap A_j' = \emptyset$ for all $i \neq j$, and
3. $\models (\mathbf{beg} \vee \mathbf{null} \vee \exists j \lceil A_j' \rceil) \wedge \lceil A_i \rceil \Rightarrow \lceil A_i' \rceil$ for all $i$.

Then by Theorem 6 we have that $\bigcap_i \mathcal{M}[\![P_i]\!]$ is an evolution condition, and hence nonempty. ∎

We now look at the Petri net semantics given in the previous chapter. Let

$$\begin{aligned} N_t &\equiv P - (O_t \cup J_t) \\ a_t &\equiv *t \wedge \bigwedge_{p \in N_t} \neg * p \wedge \bigwedge_{t' \neq t} \neg * t' \\ \sigma &\equiv \mathbf{beg} \vee \mathbf{null} \vee \bigvee_{t \in T} a_t \end{aligned}$$

We have that $\{\lceil a_t \rceil\} \cap \{\lceil a_{t'} \rceil\} = \emptyset$ for all $t \neq t'$. Furthermore,

$$\models \sigma \wedge *t \Rightarrow a_t.$$

Letting $O_t = \{p_1, \ldots, p_m\}$ and $J_t = \{p'_1, \ldots, p'_n\}$, and noting that

$$\begin{aligned} \models\ & \langle t := .t + 1, p_1 := .p_1 + 1, \ldots, p_j := .p_m + 1, p'_1 := .p'_1 - 1, \ldots, p'_k := .p'_n - 1 \rangle \Rightarrow \\ & a_t \wedge (t \uparrow \wedge \bigwedge_{p \in O_t} p \uparrow \wedge \bigwedge_{p \in J_t} p \downarrow) \end{aligned}$$

we see that $\{\lfloor \psi_t \rfloor\}$ is $(\{\lceil *t \rceil\}, \{\lceil a_t \rceil\})$-founded. Finally, it can be shown that

$$\models \sigma \Rightarrow ME \wedge \bigwedge_{p \in P} \theta_p.$$

So, by Theorem 6 we have that

$$\{\lfloor ME \wedge \bigwedge_{p \in P} \theta_p \wedge \bigwedge_{t \in T} \psi_t \rfloor\}$$

is an evolution condition. Since there is obviously a state satisfying

$$\{\lceil \bigwedge_{p \in P} (.p = M(p)) \wedge \bigwedge_{t \in T} (.t = 0) \rceil\}$$

we then have that the set of traces defining a Petri net's behavior is nonempty. ∎

# Chapter 4

# Processes

So far we have only given the semantics of two very simple programming languages. Now we look at more elaborate languages, in which the agents, which we will call *processes*, are defined using sequential constructs and recursion. As in Chapter 2, we will first present a general notation, and then define the semantics of two languages in terms of this notation. The two languages we will look at are a shared-variables language using $P$ and $V$ operations [9] for synchronization, and a variant of CSP [12] using A. J. Martin's probe function [18].

## 4.1 A language for describing sequential processes

### 4.1.1 Syntax and informal semantics

The agents defined with the notation we are about to present may be described as 'sequential processes' since they act in an essentially sequential manner, doing one thing at a time. We define a sequential process using sequential composition, IF constructs, and recursion, according to the following grammar:

$$
\begin{array}{lcl}
SP & ::= & \text{`}\alpha :\text{'}EX_2\text{`}:\text{'}BD \\
BD & ::= & CS\text{`};\textbf{end}\text{'} \\
CS & ::= & CM \mid CM\text{`};\text{'}CS \\
CM & ::= & PC \mid \text{`}[\text{'}AL\text{`}]\text{'} \mid \text{`}[\text{'}DL\text{`}|\text{'}CS\text{`}]\text{'} \\
PC & ::= & \text{`}\textbf{skip}\text{'} \mid PN \mid \text{`}\textbf{abort}\text{'} \mid \text{`}[\text{'}EX_1\text{`}\xrightarrow{\text{j}}\text{'}EX_2\text{`}]\text{'} \mid \text{`}[\text{'}EX_1\text{`}\xrightarrow{\text{f}}\text{'}EX_2\text{`}]\text{'} \\
AL & ::= & EX_1\text{`}\rightarrow\text{'}CS \mid EX_1\text{`}\rightarrow\text{'}CS\text{`}|\text{'}AL \\
DL & ::= & PN\text{`}\equiv\text{'}CS \mid PN\text{`}\equiv\text{'}CS\text{`}|\text{'}DL \\
PN & ::= & \text{procedure names} \\
EX_1 & ::= & \text{state formulas} \\
EX_2 & ::= & \text{state relations}
\end{array}
$$

where

1. A *state* formula is a local formula which contains no instance of '$\ominus$', '$\boxminus$' nor '$\lceil\rceil$'.

2. A *state relation* is a local formula containing no instance of '$\boxminus$' nor '$\lceil\rceil$' s.t., for any subexpression $\ominus\varphi$, $\varphi$ is a state formula.

Note that the value of a state formula depends only on the present state, and that of a state relation depends only on the present and immediately preceding state.

The expression '$\alpha : a : R$' defines a process with action set $\{\lceil a \rceil\}$, whose behavior is given by $R$. Informally, the meanings of the various commands are as follows: '**skip**' means 'do nothing'. '**abort**' means that something has gone wrong, and the process may do anything. '$[e \xrightarrow{\mathsf{j}} c]$' and '$[e \xrightarrow{\mathsf{f}} c]$' mean 'wait for $e$ to hold and then perform $\{\lceil c \rceil\}$ while $e$ still holds', with termination being guaranteed for the former if at some time $e$ holds and continues to hold until the action takes place, and for the latter if $e$ holds infinitely often. '**end**' means 'halt and perform no further action.'

$$[e_1 \rightarrow R_1 \mid \cdots \mid e_n \rightarrow R_n]$$

means 'wait for $e_i$ to hold for some $i$, then choose some $i$ s.t. $e_i$ holds and execute $R_i$'.

$$[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R]$$

means that $R$ is to be executed with each $p_i$ defined as $R_i$; the definitions may be mutually recursive, the various $p_i$ occurring in any of the $R_j$.

Note that we do not need to explicitly include iteration in this language, since the iteration

$$\textbf{while}\, e\, \textbf{do}\, R\, \textbf{od}$$

may be considered an abbreviation for

$$[p \equiv [e \rightarrow R \mid \neg e \rightarrow \textbf{skip}] \mid p].$$

### 4.1.2 Formal semantics

Given any $c \in EX_2$, we will write $c^{\dagger}$ for the formula obtained from $c$ by replacing any subexpression of form $\ominus\varphi$ by $\varphi$. $c^{\dagger}$ expresses a condition that holds when $\{\lceil c \rceil\}$ has just been performed, but caused no change of state; for example,

$$((.x = \ominus.y) \wedge (.y = \ominus.x))^{\dagger}$$

is equivalent to

$$.x = .y$$

and expresses the condition that the values of $x$ and $y$ were just swapped, but no state change occurred because the values were already equal. We also write $c^{\ddagger}$ for '$c \vee c^{\dagger}$'.

Given any $p \in PN$ and $R \in CS$, a free instance of $p$ in $R$ is any instance of $p$ in $R$ which is not contained within a subformula $[p_1 = R_1 \mid \cdots p_n = R_n \mid R']$ of $R$ s.t. $p = p_i$ for some $i$.

We now give the formal semantics of this notation.

**Definition:** The semantics of $SP$ is given by the semantic function

$$\mathcal{M}: SP \rightarrow \wp(Q^{\omega})$$

defined by

$$\mathcal{M}[\![\alpha : a : R]\!] \equiv \mathcal{S}_a[\![R]\!]$$

for all $a \in EX_2$ and $R \in BD$. Informally, the function

$$\mathcal{S}_a: BD \rightarrow \wp(Q^{\omega})$$

gives, for any $R \in BD$, the behavior of a process with action predicate $\{\lceil a \rceil\}$, starting at a moment when $R$ is the remainder of the program to be executed. It is defined by

- $\mathcal{S}_a[\![\mathbf{end}]\!] \equiv \{\lfloor \neg a \rfloor\}$;
- $\mathcal{S}_a[\![\mathbf{skip}; R]\!] \equiv \mathcal{S}_a[\![R]\!]$;
- $\mathcal{S}_a[\![\mathbf{abort}; R]\!] \equiv \{\lfloor \mathbf{tt} \rfloor\}$;
- $\mathcal{S}_a[\![[e \xrightarrow{\mathrm{j}} c]; R]\!] \equiv \{\lfloor \Diamond((\ominus(\boxminus\neg a \wedge e) \wedge c)^{\ddagger} \wedge \lfloor \mathcal{S}_a[\![R]\!] \rfloor) \vee (\Box\neg a \wedge \Box\diamondplus\neg e) \rfloor\}$;
- $\mathcal{S}_a[\![[e \xrightarrow{\mathrm{f}} c]; R]\!] \equiv \{\lfloor \Diamond((\ominus(\boxminus\neg a \wedge e) \wedge c)^{\ddagger} \wedge \lfloor \mathcal{S}_a[\![R]\!] \rfloor) \vee (\Box\neg a \wedge \Diamond\boxplus\neg e) \rfloor\}$;
- $\mathcal{S}_a[\![[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R']; R]\!] \equiv \mathcal{S}_a[\![R''; R]\!]$, where $R''$ is obtained from $R'$ by simultaneously replacing each free instance of any $p_i$ in $R'$ by '$[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R_i]$';
- $\mathcal{S}_a[\![p; R]\!] \equiv \{\lfloor \mathbf{tt} \rfloor\}$ if $p \in PN$;
- $\mathcal{S}_a[\![[e_1 \rightarrow R_1 \mid \cdots \mid e_n \rightarrow R_n]; R]\!] \equiv$
  $\{\lfloor \bigvee_{i=1}^{n} \Diamond(\boxminus\neg a \wedge e_i \wedge \lfloor \mathcal{S}_a[\![R_i; R]\!] \rfloor) \vee \Box(\neg a \wedge \diamondplus\neg \bigvee_{i=1}^{n} e_i) \rfloor\}$.

One might distrust this recursive definition. We previously used recursion to define $\mathcal{P}$, the function that gives the meaning of a formula. There it is obvious that, for any particular formula $\varphi$, the definition of $\mathcal{P}[\![\varphi]\!]$ can be 'unrolled' by repeated substitution to an expression which contained no instance of $\mathcal{P}$, and hence $\mathcal{P}$ is well-defined. This is not so in the recursive definition of $\mathcal{S}_a$ above; for example, one can 'unroll' the definition of

$$\mathcal{S}_a[\![[p_1 = [e \xrightarrow{\mathrm{j}} c]; p_1 \mid p_1]; R]\!]$$

forever without getting an expression that doesn't contain $\mathcal{S}_a$. One might then wonder if $\mathcal{S}_a$ is well-defined. In the next chapter we will address this issue and justify this use of recursion.

According to our definition of $\mathcal{M}$, there is no way to know that a process has reached the end of its program and terminated. If we consider it important to know this, we can have some $x \in X$ indicate whether or not the process has terminated, letting $q(x) = \mathbf{tt}$ if it has, and $q(x) = \mathbf{ff}$ otherwise, for any state $q$. We then make the last command of the program (just before the '**end**') be a command setting $x$ to $\mathbf{tt}$.

This semantics might seem ill-suited to modular reasoning about the behavior of a sequential process, since we do not give the meaning of a single command or sequence of commands, only the meaning of the 'tail end' of a complete program. However, this is no problem; we can reason about the effect of a sequence of commands $R'$ by proving that some property $\varphi(R)$ holds of $R'; R$, for all $R \in BD$. Similarly, we can reason about the effect of a command

$$[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R']$$

by first proving that, for all $R \in BD$ and $1 \leq i \leq n$, some property $\varphi_i(R)$ holds of

$$[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R_i]; R.$$

## 4.2 Shared Variables

### 4.2.1 Syntax and informal semantics

We now look at a language which describes a set of concurrent processes which interact by manipulating shared variables, with $P$ and $V$ operations on semaphores [9] for synchronization.

Let $m$ be a semaphore. When a process executes the command $P(m)$, it waits for $.m > 0$ to hold and then decrements the value of $m$ as an atomic action while $.m > 0$ still holds. If $.m > 0$ holds infinitely often, then the process is guaranteed to get its chance to complete the $P(m)$ command by decrementing the value of $m$, otherwise it may be that each time the value of $m$ is positive it is set to 0 again before the process can act. When a process executes the command $V(m)$, it increments the value of $m$. These are thus nonstrict, but fair, $P$ and $V$ operations.

Let $SEM$, $GV$ and $LV$ be pairwise disjoint sets whose elements are semaphore names, global variable names and local variable names respectively. $VN$ is the union of $GV$ and $LV$, and $G$ is the union of $GV$ and $SEM$. Letting $\mathsf{V}$ be the set of possible values of program variables, $C_P$ is a set of constant symbols denoting elements of $\mathsf{V}$, and $U_P$ (resp. $B_P$) is a set of unary (resp. binary) function symbols denoting functions with range $\mathsf{V}$. The syntax of this language is then

$$\begin{array}{lcl}
PRG & ::= & IN\,PL \\
PL & ::= & CS \mid CS\text{‘}\|\text{’}PL \\
IN & ::= & \text{‘}\mathbf{init}[\,]\text{’} \mid \text{‘}\mathbf{init}[\text{’}IL\text{‘}]\text{’} \\
IL & ::= & G\text{‘}{=}\text{’}C_P \mid G\text{‘}{=}\text{’}C_P\text{‘},\text{’}IL \\
CS & ::= & CM \mid CM\text{‘};\text{’}CS \\
CM & ::= & PC \mid \text{‘}[\text{’}AL\text{‘}]\text{’} \mid \text{‘}[\text{’}DL\text{‘}|\text{’}CS\text{‘}]\text{’} \\
PC & ::= & \text{‘}\mathbf{skip}\text{’} \mid PN \mid \text{‘}P(\text{’}SEM\text{‘})\text{’} \mid \text{‘}V(\text{’}SEM\text{‘})\text{’} \mid VN\text{‘}{\leftarrow}\text{’}EX \\
AL & ::= & EX\text{‘}{\rightarrow}\text{’}CS \mid EX\text{‘}{\rightarrow}\text{’}CS\text{‘}|\text{’}AL \\
DL & ::= & PN\text{‘}{=}\text{’}CS \mid PN\text{‘}{=}\text{’}CS\text{‘}|\text{’}DL \\
EX & ::= & EX_u \mid EX_u\,B_P\,EX \mid EX_u\text{‘}{\wedge}\text{’}EX \mid EX_u\text{‘}{=}\text{’}EX \\
EX_u & ::= & C_P \mid VN \mid U_P\,EX_u \mid \text{‘}{\neg}\text{’}EX_u \mid U_P\text{‘}(\text{’}EL\text{‘})\text{’} \mid \text{‘}(\text{’}EX\text{‘})\text{’} \\
EL & ::= & EX \mid EX\text{‘},\text{’}EL
\end{array}$$

A program is then an expression that initializes the values of several variables, followed by a number of expressions each describing a sequential process, separated by $\|$'s.

To the above syntax we add the additional restriction that for an IF command

$$[e_1 \rightarrow R_1 \mid \cdots \mid e_n \rightarrow R_n]$$

only one global variable may appear in the the guard set $\{e_1, \ldots, e_n\}$ so that the guards may be evaluated together as one atomic action. For the same reason, we also require that at most one global variable appear in an assignment statement $x \leftarrow e$, with its occurrences limited to only the right-hand side or only the left-hand side. In addition, in the initializing statement

$$\mathbf{init}[x_1 = k_1, \ldots, x_n = k_n]$$

the various $x_i$ must be distinct, and $k_i$ must denote a natural number if $x_i \in SEM$.

### 4.2.2 Formal semantics

The state space of the system described by a program is $\mathsf{V}^X$, where

$$X \equiv G \cup \{\,\chi(x) \mid x \in G\,\} \cup \{\,(x,i) \mid x \in LV \wedge i \in \mathsf{N}\,\}.$$

It is assumed that $\chi(x)$, $\chi(y)$, $x$, $y$, and $(v,i)$ are all distinct for any $x, y \in G$, $v \in LV$ and $i \in \mathsf{N}$, and that $\mathsf{V} \supseteq \mathsf{N} \cup \mathsf{B}$. Each process is identified by a unique natural number, so that $(v,i)$ is the local variable $v$ of process $i$, and for any $x \in G$, the value of $\chi(x)$ is the number of the process that last changed the value of $x$. The value of $\chi(x)$ is meaningless if the value of $x$ has never changed. $\{\![a_i]\!\}$ is the action predicate of process $i$, where

$$a_i \equiv \exists x \in G(*x \wedge .\chi(x) = i) \vee \exists x \in LV(*(x, i)).$$

The meaning of a program

$$\mathbf{init}[x_1 = e_1, \cdots, x_n = e_n]\, P_1 \parallel \cdots \parallel P_k$$

is then

$$\{\![(\mathbf{beg} \Rightarrow \bigwedge_{i=1}^{n} (.x_i = e_i)) \wedge \varphi_G]\!\} \cap \bigcap_{i=1}^{k} T_i$$

where

$$\begin{aligned} \varphi_G &\equiv \forall m \in SEM(.m \in \mathbb{N}) \wedge \forall x \in G(.\chi(x) \in \{1, \ldots, k\}) \\ T_i &\equiv \mathcal{M}[\![\alpha : a_i : P_i'; \mathbf{end}]\!] \end{aligned}$$

and for all $i$, $P_i'$ is obtained from $P_i$ as follows:

First, every instance of an $x \in GV$ not on the left-hand side of an assignment is replaced by '$.x$', and every instance of a $y \in LV$ not on the left-hand side of an assignment is replaced by '$.(y, i)$'. Second, every local assignment

$$x \leftarrow e \qquad (\text{where } x \in LV)$$

is replaced by

$$[\mathbf{tt} \xrightarrow{\mathrm{j}} .(x, i) = \ominus e \wedge \rho(x)]$$

where for all $x' \in VN$,

$$\rho(x') \equiv \forall y \in G(.\chi(y) = i \wedge *y \Rightarrow y = x') \wedge \forall y \in LV(*(y, i) \Rightarrow y = x')$$

(this says that process $i$ changes only the value of $x'$). Third, every global assignment

$$x \leftarrow e \qquad (\text{where } x \in GV)$$

is replaced by

$$[\mathbf{tt} \xrightarrow{\mathrm{j}} .\chi(x) = i \wedge .x = \ominus e \wedge \rho(x)].$$

Finally, every command $P(m)$ is replaced by

$$[.m > 0 \xrightarrow{\mathrm{f}} .\chi(m) = i \wedge .m = \ominus .m - 1 \wedge \rho(m)]$$

and every command $V(m)$ is replaced by

$$[\mathbf{tt} \xrightarrow{\mathrm{j}} .\chi(m) = i \wedge .m = \ominus .m + 1 \wedge \rho(m)].$$

Here we see the utility of the notation we introduced in the previous section: each $P_i$ may be regarded as simply an abbreviation for an expression in $SP$. We will take the same approach in defining the semantics of the language presented in the next section.

### 4.2.3 Satisfiability

Let us rewrite

$$T: \qquad \{\lfloor(\mathbf{beg} \Rightarrow \bigwedge_{i=1}^{n}(.x_i = e_i)) \wedge \varphi_G\rfloor\} \cap \bigcap_{i=1}^{k} T_i$$

as

$$\{\lfloor(\mathbf{beg} \Rightarrow \varphi_G \wedge \bigwedge_{i=1}^{n}(.x_i = e_i)) \wedge (\ominus\varphi_G \Rightarrow \varphi_G)\rfloor\} \cap \bigcap_{i=1}^{k} T_i.$$

We can do this because, in general,

$$\models \Box\varphi \Leftrightarrow \Box((\mathbf{beg} \Rightarrow \varphi) \wedge (\ominus\varphi \Rightarrow \varphi)).$$

For all $i$, let

$$\begin{aligned} a'_i \;\equiv\; & \exists x \in LV, v \in \mathsf{V}(*x \wedge x := v) \vee \\ & \exists x \in G\, \exists v \in \mathsf{V}(*x \wedge \langle x := v, \chi(x) := i\rangle \wedge (x \in SEM \wedge \ominus.x \in \mathsf{N} \Rightarrow v \in \mathsf{N})). \end{aligned}$$

It is straightforward to check that

1. $\models \neg(a'_i \wedge a'_j)$ for all $i \neq j$,
2. $\models (\mathbf{beg} \vee \mathbf{null} \vee \exists j(a'_j)) \wedge a_i \Rightarrow a'_i$ for all $i$, and
3. $\models (\mathbf{beg} \vee \mathbf{null} \vee \exists j(a'_j)) \Rightarrow (\ominus\varphi_G \Rightarrow \varphi_G)$,

and so applying Theorem 6 with

1. $A_i \equiv \{\lceil a_i\rceil\}$, $A'_i \equiv \{\lceil a'_i\rceil\}$, and
2. $T'_G \equiv \{\lfloor\ominus\varphi_G \Rightarrow \varphi_G\rfloor\}$

we see that if $\mathcal{M}[\![\alpha : a_i : P'_i; \mathbf{end}]\!]$ is $(A_i, A'_i)$-founded for all $i$, then

$$\{\lfloor(\ominus\varphi_G \Rightarrow \varphi_G) \wedge \bigwedge_{i=1}^{k} \varphi_i\rfloor\}$$

is an evolution condition. Given the restrictions we placed on the initializing statement, it is easily seen that there is a state satisfying

$$\varphi_G \wedge \bigwedge_{i=1}^{n}(.x_i = e_i),$$

and hence the trace set $T$ is nonempty.

It remains only to show that $\mathcal{M}[\![\alpha : a_i : P'_i; \mathbf{end}]\!]$ is $(A_i, A'_i)$-founded for all $i$. The proof of this requires a result, Theorem 12, which will be proven in the next chapter. Theorem 12 states that, for any $R \in BD$ and state relations $a$ and $a'$, if

1. $\{\lceil a\rceil\}$ and $\{\lceil a'\rceil\}$ are action predicates and $\models a' \Rightarrow a$, and
2. for every command of form $[g \xrightarrow{\mathrm{j}} c]$ or $[g \xrightarrow{\mathrm{f}} c]$ in $R$ we can find some state relation $c'$ s.t. $\models c' \Rightarrow c$ and $(\{\lceil g\rceil\}, \{\lceil c'\rceil\})$ is $\{\lceil a'\rceil\}$-admissible

then $\mathcal{M}[\![\alpha : a_i : R]\!]$ is $(\{\lceil a\rceil\}, \{\lceil a'\rceil\})$-admissible. We apply this result to $P'_i$:

1. $\models a'_i \Rightarrow a_i$, hence $A'_i \subseteq A_i$.

2. For the assignment '$[\mathbf{tt} \xrightarrow{\mathrm{j}} c]$', where
$$c \equiv .\chi(x) = i \wedge .x = \ominus e \wedge \rho(x)$$
($x \in GV$), we choose '$\langle x := e, \chi(x) := i\rangle$' for $c'$, since
$$\models \langle x := e, \chi(x) := i\rangle \Rightarrow c \wedge (a'_i \vee \mathbf{null}).$$

3. For the $P$ action '$[.m > 0 \xrightarrow{\mathrm{f}} c]$', where
$$c \equiv .\chi(m) = i \wedge .m = \ominus .m - 1 \wedge \rho(m),$$
we choose '$\langle m := .m - 1, \chi(m) := i\rangle$' for $c'$, since
$$\models \langle m := .m - 1, \chi(m) := i\rangle \Rightarrow c$$
and
$$\models \langle m := .m - 1, \chi(m) := i\rangle \wedge \ominus(.m > 0) \Rightarrow a'_i \vee \mathbf{null}.$$

4. Similarly for local assignments and $V$ actions.

Hence $\mathcal{M}[\![\alpha : a_i : P'_i; \mathbf{end}]\!]$ is $(A_i, A'_i)$-founded. ∎

## 4.3 Communicating Sequential Processes

### 4.3.1 Syntax and informal semantics

As our second example we look at a modification of Hoare's "Communicating Sequential Processes", or CSP [12], described by A. J. Martin in [18]. It is the same as regular CSP except that

- communication actions never appear in guards;
- there is an extra primitive, a function called the *probe*.

In this language processes interact, not by modifying the values of a pool of shared variables, but only by sending values to each other. Special *communications actions* are introduced: '$e_1!e_2$' means 'send the value $e_2$ to process $e_1$', and '$e?x$' means 'receive a value from process $e$ and store it in the (internal) variable $x$'. There is no automatic buffering, and hence the communication actions are tightly coupled. If process $i$ wants to send a value to process $j$, it must wait at its send action until process $j$ comes to a matching receive action, and then the communication takes place. Similarly, if process $i$ wants to receive a value from process $j$, it must wait at the receive action until process $j$ comes to a matching send action.

We say that a process is suspended when it is waiting at a communication action. The probe is used to test whether or not a process may become suspended if it initiates a communication. For a process $i$, if the probe $\mathrm{ps}\, j$ is true this means that process $j$ is suspended at a receive from $i$, i.e. it is waiting to receive a value from $i$, and hence process $i$ will not become suspended if it tries to send a message to $j$; conversely, if process $j$ is suspended at a receive from $i$ then $\mathrm{ps}\, j$ will eventually become true unless $j$ is released from suspension first, by $i$ sending it a value. Similarly, if the probe $\mathbf{pr}\,j$ is true then process $j$ is suspended at a send to $i$ and hence process $i$ will not become suspended if it tries to receive a message from $j$, etc. Once $\mathbf{ps}\,j$ becomes true it remains true until the process sends a message to $j$ (thus releasing $j$ from suspension), and similarly for $\mathbf{pr}\,j$.

In an implementation of a CSP program, the probe values may be defined in any way consistent with the above description. In particular, we can define the value of any of a process' probes to remain constant during the evaluation of an expression containing that probe, after the appropriate external values have been sampled. If we do this, then the evaluation of an expression containing several different probes can be considered an atomic action, since each probe value, once determined, remains constant throughout the remainder of the expression's evaluation.

The syntax of the language is similar to that of the previous section, and hence we give here only the grammar rules that differ from what was presented there:

$$\begin{array}{lll}
PRG & ::= & PRC \mid PRC`\|'PRG \\
PRC & ::= & NAT`\text{:}'CS \\
PC & ::= & `\mathbf{skip}' \mid PN \mid EX`!'EX \mid EX`?'LV \mid LV`\leftarrow'EX \\
EX_u & ::= & C_P \mid LV \mid U_P\, EX_u \mid `\neg'EX_u \mid U_P`('EL`)' \mid `('EX`)' \mid PR \\
PR & ::= & `\mathbf{ps}'EX_u \mid `\mathbf{pr}'EX_u \\
NAT & ::= & \text{constant symbols denoting natural numbers} \\
 & \vdots &
\end{array}$$

We also require for a program $n_1 : c_1 \| \cdots \| n_k : c_k$ that $n_1, \ldots, n_k$ all be distinct, and that no probe appear in a command of form $e_1!e_2$ or $e?x$.

### 4.3.2 Formal semantics

As in the previous section, $\mathsf{V}$ will be the set of possible values of program variables, and we assume that $\mathsf{V} \supseteq \mathsf{N} \cup \mathsf{B}$. The state space of the system described by a program is $(\mathsf{V}^*)^X$, where $\mathsf{V}^*$ is the set of finite (and possibly empty) sequences of elements of $\mathsf{V}$, and

$$X \equiv \{\, (x,i), \mathbf{s}(i,j), \mathbf{r}(i,j), \mathbf{ps}(i,j), \mathbf{pr}(i,j) \mid x \in LV \wedge i,j \in \mathsf{N} \,\}.$$

For all $x \in LV$, $i,j,i',j' \in \mathsf{N}$ and $f,g \in \{\mathbf{s},\mathbf{r},\mathbf{ps},\mathbf{pr}\}$ we have that $f(i,j) \neq (x,i')$, and if $f(i,j) = g(i',j')$ then $f = g$, $i = i'$ and $j = j'$. We equate the sequence of length one formed from $v \in \mathsf{V}$ with $v$ itself, and so $\mathsf{V} \subseteq \mathsf{V}^*$. In addition, we define

$$X_i \equiv \{\, (x,i), \mathbf{s}(i,j), \mathbf{r}(j,i) \mid x \in LV \wedge j \in \mathsf{N} \,\}.$$

$\mathbf{s}(i,j)$ indicates the sequence of values that process $i$ has sent or tried to send to $j$; if $i$ is suspended at a send to $j$, then the last element of this sequence is the value that it is trying to send. $\mathbf{r}(i,j)$ indicates the number of times that process $j$ has initiated the receipt of a value from $i$; if $j$ is suspended waiting to receive a value from $j$, then this will be one more than the number of values actually received so far from $j$. $\mathbf{pr}(i,j)$ indicates process $j$'s probe to see if $i$ is sending it a value, and $\mathbf{ps}(i,j)$ is process $i$'s probe to see if $j$ is ready to receive a value.

We introduce the abbreviations

$$\begin{array}{lll}
\mathbf{qs}(i,j) & \equiv & \ell(.\mathbf{s}(i,j)) > .\mathbf{r}(i,j) \\
\mathbf{qr}(i,j) & \equiv & .\mathbf{r}(i,j) > \ell(.\mathbf{s}(i,j))
\end{array}$$

where $\ell(x)$ is the length of sequence $x$. $\mathbf{qs}(i,j)$ holds when process $i$ is suspended trying to send to $j$, and $\mathbf{qr}(i,j)$ holds when process $j$ is suspended trying to receive from $i$. We will also write '$ext(x,a)$' for sequence $x$ extended by one element, $a$, and '$el(n,x)$' for the $n$-th element of sequence $x$.

The following hold at all time:

A0: $\forall i,j \in \mathsf{N}(.\mathbf{r}(i,j) \in \mathsf{N} \wedge .\mathbf{s}(i,j) \in \mathsf{V}^* \wedge .\mathbf{pr}(i,j) \in \mathsf{B} \wedge .\mathbf{ps}(i,j) \in \mathsf{B})$

A1: $\forall i,j \in \mathsf{N}(*\mathbf{s}(i,j) \Rightarrow \exists v \in \mathsf{V}(.\mathbf{s}(i,j) = ext(\ominus.\mathbf{s}(i,j),v)))$

A2: $\forall i,j \in \mathsf{N}(*\mathbf{r}(i,j) \Rightarrow .\mathbf{r}(i,j) = \ominus.\mathbf{r}(i,j)+1)$

A3: $\forall i,j \in \mathsf{N}(-1 \le \ell(.\mathbf{s}(i,j)) - .\mathbf{r}(i,j) \le 1)$

A1 says that the value of $\mathbf{s}(i,j)$ can only be changed by extending it with a new element. A2 says that the value of $\mathbf{r}(i,j)$ can only be changed by incrementing it. Given the definitions of $\mathbf{qs}$ and $\mathbf{qr}$, A3 implies that process $i$ cannot initiate another send to $j$ while it is suspended at a send to $j$, and process $j$ cannot initiate another receive from $i$ while it is suspended at a receive from $i$.

A4: $\forall i,j \in \mathsf{N}(.\mathbf{ps}(i,j) \Rightarrow \mathbf{qr}(i,j))$

A5: $\forall i,j \in \mathsf{N}(.\mathbf{pr}(i,j) \Rightarrow \mathbf{qs}(i,j))$

A6: $\forall i,j \in \mathsf{N}(\neg.\mathbf{pr}(i,j) \wedge *\mathbf{pr}(i,j) \Rightarrow \neg\mathbf{qs}(i,j))$

A7: $\forall i,j \in \mathsf{N}(\neg.\mathbf{ps}(i,j) \wedge *\mathbf{ps}(i,j) \Rightarrow \neg\mathbf{qr}(i,j))$

A8: $\forall i,j \in \mathsf{N}(\mathbf{qr}(i,j) \Rightarrow \oplus(.\mathbf{ps}(i,j) \vee \neg\mathbf{qr}(i,j)))$

A9: $\forall i,j \in \mathsf{N}(\mathbf{qs}(i,j) \Rightarrow \oplus(.\mathbf{pr}(i,j) \vee \neg\mathbf{qs}(i,j)))$

These formalize the description of the probe we gave earlier.

Initially, the following holds:

A-in: $\forall i,j \in \mathsf{N}(.\mathbf{r}(i,j) = 0 \wedge .\mathbf{s}(i,j) = \varepsilon \wedge \neg.\mathbf{pr}(i,j) \wedge \neg.\mathbf{ps}(i,j))$,

i.e. no receives nor sends have been initiated, and the probes are all false.

The meaning of a program $n_1 : P_1 \parallel \cdots \parallel n_k : P_k$ is then

$$\{\lfloor(\text{A0–A9}) \wedge (\mathbf{beg} \Rightarrow \text{A-in})\rfloor\} \cap \bigcap_{i=1}^{k} T_i$$

where for all $i$,

$$\begin{aligned} T_i &\equiv \mathcal{M}[\![\alpha : a_i : P_i'; \mathbf{end}]\!] \\ a_i &\equiv \exists x \in X_{n_i}(*x) \end{aligned}$$

and $P_i'$ is obtained from $P_i$ as follows:

First, every subexpression of form $\mathbf{ps}\, e$ is replaced by '$.\mathbf{ps}(n_i,e)$', and every subexpression of form $\mathbf{pr}\, e$ is replaced by '$.\mathbf{pr}(e,n_i)$'. Second, every instance of a program variable $y \in LV$ which is not the left-hand side of an assigment $y \leftarrow e$ or the right-hand side of a receive command $e!y$ is replaced by '$.(y, n_i)$'. Third, every instance of a send command $e_1!e_2$ is replaced by

$$\begin{array}{lll} [ & e_1 \notin \mathsf{N} \rightarrow & \textbf{abort} \\ | & e_1 \in \mathsf{N} \rightarrow & [\neg \mathbf{qs}(n_i, e_1) \xrightarrow{\mathsf{j}} \mathit{initsend}_i]; \\ & & [\neg \mathbf{qs}(n_i, e_1) \rightarrow \textbf{skip}] \\ ] & & \end{array}$$

where

$$\begin{array}{lll} \mathit{initsend}_i & \equiv & \rho(\mathrm{s}(n_i, e_1)) \wedge .\mathrm{s}(n_i, e_1) = \mathit{ext}(\ominus.\mathrm{s}(n_i, e_1), e_2) \\ \rho(x) & \equiv & \forall x' \in X_{n_i}(*x' \Rightarrow x' = x) \qquad \text{for all } x \in X_{n_i}. \end{array}$$

The expression $\mathit{initsend}_i$ just says that process $n_i$ extends the value of $\mathrm{s}(n_i, e_1)$ with $e_2$, and causes no other state change. Fourth, every instance of a receive command $e?y$ is replaced by

$$\begin{array}{lll} [ & e \notin \mathsf{N} \rightarrow & \textbf{abort} \\ | & e \in \mathsf{N} \rightarrow & [\neg \mathbf{qs}(e, n_i) \xrightarrow{\mathsf{j}} \mathit{initrecv}_i]; \\ & & [\neg \mathbf{qr}(e, n_i) \rightarrow y \leftarrow \mathit{el}(.\mathbf{r}(e, n_i), .\mathrm{s}(e, n_i))] \\ ] & & \end{array}$$

where

$$\mathit{initrecv}_i \equiv \rho(\mathbf{r}(e, n_i)) \wedge .\mathbf{r}(e, n_i) = \ominus.\mathbf{r}(e, n_i) + 1.$$

The expression $\mathit{initrecv}_i$ just says that process $n_i$ increments the value of $\mathbf{r}(e, n_i)$, and causes no other state change. Finally, every command of form $y \leftarrow e$ is replaced by

$$[\mathbf{tt} \xrightarrow{\mathsf{j}} .(y, n_i) = \ominus e \wedge \rho(y, n_i)].$$

The purpose of the guard '$\neg \mathbf{qs}(n_i, e_1)$' in the command

$$[\neg \mathbf{qs}(n_i, e_1) \xrightarrow{\mathsf{j}} \mathit{initsend}_i]$$

used above is to ensure that A3 is maintained (similarly for a receive action). It seems certain that this guard could be replaced by '$\mathbf{tt}$' without changing the semantics, since process $n_i$ must wait for $\mathbf{qs}(n_i, e_1)$ to go false before leaving the send command, and this can only become true again by process $n_i$ initiating a send to $e_1$. However, at the time of this writing the author has not proven this assertion, hence we keep the more complex guard.

### 4.3.3 Satisfiability

For all $i, j \in \mathsf{N}$, let

$$\begin{array}{lll} a^!_i & \equiv & \exists x \in LV(*(x, i)) \\ & & \vee \exists j \in \mathsf{N}(\ominus \neg \mathbf{qr}(j, i) \wedge \langle \mathbf{r}(j, i) := .\mathbf{r}(j, i) + 1, \mathbf{pr}(j, i) := \mathbf{ff} \rangle) \\ & & \vee \exists j \in \mathsf{N}, v \in \mathsf{V}(\ominus \neg \mathbf{qs}(i, j) \wedge \langle \mathrm{s}(i, j) := \mathit{ext}(.\mathrm{s}(i, j), v), \mathbf{ps}(i, j) := \mathbf{ff} \rangle) \\ a^!_{ij} & \equiv & \neg \mathbf{null} \wedge ((\ominus \mathbf{qr}(i, j) \wedge \langle \mathbf{ps}(i, j) := \mathbf{tt} \rangle) \vee (\ominus \mathbf{qs}(i, j) \wedge \langle \mathbf{pr}(i, j) := \mathbf{tt} \rangle)) \\ \varphi_{ij} & \equiv & [\mathbf{qs}(i, j) \xrightarrow{\mathsf{j}} \mathbf{pr}(i, j) := \mathbf{tt} \mid \mathbf{qr}(i, j) \xrightarrow{\mathsf{j}} \mathbf{ps}(i, j) := \mathbf{tt} \mid \alpha : a^!_{ij} \mid \varepsilon : \mathbf{ff}] \end{array}$$

Noting that

$$\models \varphi_{ij} \Rightarrow \text{A8} \wedge \text{A9},$$

we see that

$$\{\lfloor(\mathbf{beg} \Rightarrow \text{A-in} \wedge AX) \wedge (\ominus AX \Rightarrow AX) \wedge \forall i,j \in \mathsf{N}(\varphi_{ij})\rfloor\} \cap \bigcap_{i=1}^{k} T_i \subseteq T,$$

where $AX \equiv$ (A0–A7).

Defining $a_{ij} \equiv a'_{ij}$, it is straightforward to check that

1. $\models \neg(a'_i \wedge a'_j)$ for all $i \neq j$,
2. $\models \neg(a'_i \wedge a'_{kl})$ for all $i, k, l$,
3. $\models \neg(a'_{ij} \wedge a'_{kl})$ for all $(i,j) \neq (k,l)$,
4. $\models (\mathbf{beg} \vee \mathbf{null} \vee \exists j(a'_j) \vee \exists k,l(a'_{kl})) \wedge a_i \Rightarrow a'_i$ for all $i$,
5. $\models (\mathbf{beg} \vee \mathbf{null} \vee \exists j(a'_j) \vee \exists k,l(a'_{kl})) \Rightarrow (\ominus AX \Rightarrow AX)$, and
6. $\{\lfloor\varphi_{ij}\rfloor\}$ is $(\{\lceil a_{ij}\rceil\}, \{\lceil a'_{ij}\rceil\})$-founded for all $i$ and $j$,

and so applying Theorem 6 with

1. $A_i \equiv \{\lceil a_i\rceil\}$, $A'_i \equiv \{\lceil a'_i\rceil\}$, $A_{ij} \equiv A'_{ij} \equiv \{\lceil a'_{ij}\rceil\}$,
2. $T'_G \equiv \{\lfloor \ominus AX \Rightarrow AX\rfloor\}$,
3. $T_{ij} \equiv \{\lfloor\varphi_{ij}\rfloor\}$,

we see that if $T_i = \mathcal{M}[\![\alpha : a_i : P'_i; \mathbf{end}]\!]$ is $(A_i, A'_i)$-founded for all $i$, then

$$\{\lfloor(\mathbf{beg} \Rightarrow \text{A-in} \wedge AX) \wedge (\ominus AX \Rightarrow AX) \wedge \forall i,j \in \mathsf{N}(\varphi_{ij})\rfloor\} \cap \bigcap_{i=1}^{k} T_i$$

is an evolution condition. Since it is easily seen that there is a state satisfying A-in and A0–A7, we then have that trace set $T$ is nonempty.

It remains only to show that $\mathcal{M}\alpha : a_i : P'_i; \mathbf{end}$ is $(A_i, A'_i)$-founded. This can be done using Theorem 12. It is straightforward to verify, as we did with the shared-variables language, that the conditions for applying this theorem hold, hence we omit the details. ∎

# Chapter 5

# Recursion

We now address some questions which we deferred in the previous chapter. We will discuss how the recursive definition of $S_a$ in the previous chapter and such definitions in general are to be interpreted, and how to prove satisfiability for programming languages using recursion.

## 5.1 Recursion and Domains

In general, recursive definitions are of the form $x = E$, where $E$ is an expression containing $x$; this is equivalent to $x = F(x)$, where $F$ is the function $\lambda x.E$. A solution to the equation $x = F(x)$ is called a *fixed-point* of $F$. In general, $F$ may have zero, one, or many fixed-points, and hence the equation $x = F(x)$ may not define anything or may not define $x$ uniquely. It is sometimes the case, as in the definition of $\mathcal{P}$ in Chapter 2, that a recursively-defined function $f$ is recursively defined on the structure of its argument. In such a case we can, for any argument $x$, 'unroll' the definition of $f(x)$ by repeated substitution using the definition of $f$, and so obtain an expression for $f(x)$ which does not contain any instance of $f$; hence the function is well-defined. However, only a restricted class of recursive definitions can be 'unrolled' this way, and so we need more general method of guaranteeing that our recursive definitions do indeed uniquely define some object. This is commonly done by the use of ***domains***.[1]

**Definition:** A ***domain*** is a set $\mathcal{D}$ with a partial order $\sqsubseteq$ s.t. every countable, totally ordered $C \subseteq \mathcal{D}$ has a least upper bound in $\mathcal{D}$, denoted $\bigsqcup C$. In particular, the empty set has a least upper bound in $\mathcal{D}$, i.e. there is an element $\perp \in \mathcal{D}$ s.t. $\perp \sqsubseteq x$ for all $x \in \mathcal{D}$. A ***strong domain*** is a domain $\mathcal{D}$ s.t. *every* totally ordered $C \subseteq \mathcal{D}$ has a least upper bound in $\mathcal{D}$.

A totally ordered subset of a domain $\mathcal{D}$ is called a ***chain*** in $\mathcal{D}$. We will often index the elements of a chain $C$ by some totally ordered set $J$, letting $C = \{ x_i \mid i \in J \}$, where $i \leq j \Rightarrow x_i \sqsubseteq x_j$ for all $i, j \in J$, and writing $\bigsqcup_i x_i$ for $\bigsqcup C$. The partial ordering $\sqsubseteq$ of a domain we call its ***approximation ordering***, since $x \sqsubseteq y$ is intended to mean that, in some sense, $x$ is an approximation of $y$.

When necessary to distinguish between the bottom elements of different domains, we will write $\perp_{\mathcal{D}}$ for the bottom element of domain $\mathcal{D}$.

One example of a strong domain is $\wp(Y)$, the powerset of some set $Y$. If we take the inclusion relation $\subseteq$ as its approximation ordering, then $\perp = \emptyset$ and $\bigsqcup_i x_i = \bigcup_i x_i$; alternatively, if we take $\supseteq$ as its approximation ordering, then $\perp = Y$ and $\bigsqcup_i x_i = \bigcap_i x_i$. From now on, when we refer to the domain $\wp(Y)$ for some set $Y$, the latter approximation ordering ($\supseteq$) will be assumed.

[1]The properties mentioned in this section are all well known, and hence will not be proven. See, for example, [16].

For any set $Y$ and domain (resp. strong domain) $\mathcal{D}$, $\mathcal{D}^Y$ is also a domain (resp. strong domain), with approximation ordering $\sqsubseteq$ given by

$$f_1 \sqsubseteq f_2 \equiv \forall y \in Y (f_1(y) \sqsubseteq f_2(y))$$

for any $f_1, f_2 \colon Y \to \mathcal{D}$. For any $y \in Y$ and chain $\{f_i\}$ in $\mathcal{D}^Y$, we have that $(\bigsqcup_i f_i)(y) = \bigsqcup_i f_i(y)$. Similarly, given domains (resp. strong domains) $\mathcal{D}_1, \ldots, \mathcal{D}_n$, the cartesian product $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ is also a domain (resp. strong domain), with approximation ordering $\sqsubseteq$ given by

$$(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \equiv x_1 \sqsubseteq y_1 \wedge \cdots \wedge x_n \sqsubseteq y_n$$

for any $\bar{x}, \bar{y} \in \mathcal{D}$. For any chain $\{\bar{x}_i\}$ in $\mathcal{D}$ we have that $\bigsqcup_i \bar{x}_i = (\bigsqcup_i x_{i,1}, \ldots, \bigsqcup_i x_{i,n})$.

**Definition:** Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two domains; then a function $f \colon \mathcal{D}_1 \to \mathcal{D}_2$ is *monotonic* iff $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ for all $x, y \in \mathcal{D}_1$, and it is *continuous* iff it is monotonic and $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$ for any countable chain $\{x_i\}$ in $\mathcal{D}_1$.

For any two domains (resp. strong domains) $\mathcal{D}_1$ and $\mathcal{D}_2$, the set of monotonic functions from $\mathcal{D}_1$ to $\mathcal{D}_2$, denoted $(\mathcal{D}_1 \overset{\mathrm{m}}{\to} \mathcal{D}_2)$, is also a domain (resp. strong domain), as is the set of continuous functions, denoted $(\mathcal{D}_1 \overset{\mathrm{c}}{\to} \mathcal{D}_2)$, with the same approximation ordering as the domain $\mathcal{D}_2^{\mathcal{D}_1}$.

In order to show that a function $f \colon \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to \mathcal{D}$ is monotonic, it suffices to show that $f$ is monotonic in each argument, i.e. given any $1 \leq j \leq n$ and $\bar{y} \in \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$, we have that

$$f(y_1, \ldots, y_{j-1}, x, y_{j+1}, \ldots, y_n) \sqsubseteq f(y_1, \ldots, y_{j-1}, x', y_{j+1}, \ldots, y_n)$$

for any $x, x' \in \mathcal{D}_j$ s.t. $x \sqsubseteq x'$. To show that $f$ is continuous it also suffices to show that $f$ is continuous in each argument, i.e. $f$ is monotonic in each argument and, given any $j$, $\bar{y}$ and countable chain $\{x_i\}$ in $\mathcal{D}_j$,

$$f(y_1, \ldots, y_{j-1}, \bigsqcup_i x_i, y_{j+1}, \ldots, y_n) = \bigsqcup_i f(y_1, \ldots, y_{j-1}, x_i, y_{j+1}, \ldots, y_n).$$

**Definition:** Given a domain $\mathcal{D}$, a *fixed-point* of a function $F \colon \mathcal{D} \to \mathcal{D}$ is any $x \in \mathcal{D}$ s.t. $x = F(x)$. A *least fixed-point* of $F$ is a fixed-point $x$ of $F$ s.t. $x \sqsubseteq y$ for all fixed-points $y$ of $F$; if such an $x$ exists it is unique, and is denoted *fix*$(F)$.

It has been shown that for any strong domain $\mathcal{D}$ and monotonic function $F \colon \mathcal{D} \overset{\mathrm{m}}{\to} \mathcal{D}$, $F$ has a least fixed-point. In fact, *fix*$(F) = \gamma_i$ for some ordinal[2] $i$, where $\gamma_0 \equiv \bot$, $\gamma_{k+1} \equiv F(\gamma_k)$ for any ordinal $k$, and $\gamma_k \equiv \bigsqcup_{j<k} \gamma_j$ for any limit ordinal[3] $k$ (note that $\gamma_j \sqsubseteq \gamma_k$ for all $j \leq k$).[4] In addition, it has been shown that for any domain $\mathcal{D}$ and continuous function $F \colon \mathcal{D} \overset{\mathrm{c}}{\to} \mathcal{D}$, $F$ has a least fixed-point, viz. $\gamma_\omega$.

Given domains $\mathcal{D}_i$ ($1 \leq i \leq n+1$) and any set $Y$, the following are some continuous (resp. monotonic) functions:

[2]The ordinals are a totally ordered extension of the natural numbers s.t. every ordinal $i$ has a successor, $i+1$, which is the least ordinal greater than $i$, and any set of ordinals has a least upper bound which is also an ordinal. See any text on set theory for further details.

[3]A limit ordinal is a nonzero ordinal which is not the successor of any other ordinal. An example is $\omega$, the least upper bound of the natural numbers.

[4]A proof of these may be found in [20] for the strong domain $\wp(Y)$ with approximation ordering $\subseteq$, for any set $Y$; the proof is easily generalized to any strong domain.

- any constant function $f: \mathcal{D}_1 \to \mathcal{D}_2$;
- the identity function on $\mathcal{D}_1$;
- the function $\lambda f.\lambda y.f(h(y)): \mathcal{D}_1^Y \to \mathcal{D}_1^Y$ for any function $h: Y \to Y$.
- the function $\lambda f.f(y): \mathcal{D}_1^Y \to \mathcal{D}_1$ for any $y \in Y$;
- any function $f: \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to \mathcal{D}_{n+1}$ defined by $f(x_1, \ldots, x_n) = e$, where $e$ contains only constants, continuous (resp. monotonic) functions and the variables $x_1, \ldots, x_n$.

The recursive definition $x \equiv E$, where $E$ is an expression constructed from $x$, constants and continuous (resp. monotonic) functions, is then taken to be an abbreviation for $x \equiv \mathit{fix}(F)$, where $F$ is defined by $F(x) \equiv E$, and similarly for a recursive function definition $f(x) \equiv E'$, where $E'$ may contain instances of both $f$ and $x$. For example, the recursive definition of $\mathcal{S}_a$ given in the previous chapter may be considered an abbreviation for $\mathcal{S}_a \equiv \mathit{fix}(F)$, where

$$F: BD \to \wp(Q^\omega)$$

is a monotonic function defined as follows:

- $F(f)[\![\mathbf{end}]\!] \equiv \{\lfloor \neg a \rfloor\}$;
- $F(f)[\![\mathbf{skip}; R]\!] \equiv f[\![R]\!]$;
- $F(f)[\![\mathbf{abort}]\!] \equiv \{\lfloor \mathbf{tt} \rfloor\}$;

$\vdots$

An inspection of the definition of $\mathcal{S}_a$ will reveal that $F$ is indeed monotonic, and so we have justified our recursive definition of $\mathcal{S}_a$.

## 5.2 Core domains

### 5.2.1 Motivation

In the previous chapter we were interested in proving that some trace set was $(A, A')$-founded for appropriate action predicates $A$ and $A'$. The usual way to show that some object $x$ defined by $x \equiv F(x)$ has a certain property is to show that the set of things having the property forms a domain (or strong domain), and that $F$ is a continuous (resp. monotonic) function on this domain. For our application, the natural approximation ordering on $Q^\omega$ to use is the superset ordering

$$T_1 \sqsubseteq T_2 \equiv T_1 \supseteq T_2,$$

i.e. $T_1$ approximates $T_2$ iff $T_1$ is a weaker constraint on the system's behavior than $T_2$, which makes $Q^\omega$ a strong domain. Unfortunately, the set of $(A, A')$-founded trace sets does not form a subdomain of $\wp(Q^\omega)$ under this ordering, as the following example shows: Let

1. $A \equiv A' \equiv \{\lceil .n > \ominus.n \rceil\}$, and
2. $T_i \equiv G(\{l_i\})$ for all $i$, where $l_i \equiv (\{\lceil \mathbf{tt} \rceil\}, \{\lceil .n > \ominus.n + i \rceil\})$.

It is then easily seen that $T_i$ is $(A, A')$-founded and $T_i \sqsubseteq T_j$ ($T_i \supseteq T_j$) for all $i \leq j$, but $\bigsqcup_i T_i = \bigcap_i T_i = \emptyset$, and the empty set is definitely not $(A, A')$-founded.

Thus, for example, given action predicates $A$ and $A'$ and a monotonic function $f$ on trace sets which maps $(A, A')$-founded trace sets to $(A, A')$-founded trace sets, we cannot guarantee *a priori* that $\mathit{fix}(f)$ is also $(A, A')$-founded. Similar problems arise when we take the fixed-points of monotonic functions on strong domains derived from $\wp(Q^\omega)$, in particular when we define $S_a \equiv \mathit{fix}(F)$, where $F$ is the function on the domain $\wp(Q^\omega)^{BD}$ defined at the end of the previous section.

As we shall see, these problems can be overcome through the use of *core domains* which in some sense parallel the strong domains we are interested in. We begin by defining a family of domains from which all our core domains will be derived.

### 5.2.2 Definitions

**Definition:** Given action predicates $A$ and $A'$ s.t. $A' \subseteq A$, $C(A, A')$ is the set of tuples $(S, L)$ s.t. $S \subseteq Q^+$, $L$ is a countable set of $A'$-admissible transition rules and

$$\mathit{safe}(L, A) \subseteq \{\!\lfloor \lceil S \rceil \rfloor\!\}.$$

For any $(S, L), (S', L') \in C(A, A')$ we define

$$(S, L) \sqsubseteq (S', L') \equiv S \supseteq S' \wedge L \subseteq L'.$$

**Property 7** *If $(S, L) \in C(A, A')$ then $H(L, A) \subseteq S$, where for any set $L$ of transition rules and action predicate $A$,*

$$H(L, A) \equiv \{\!\lceil \boxminus(\neg\lceil A \rceil \vee \exists(\pi, \tau) \in L'(\lceil \tau \rceil \wedge \ominus\lceil \pi \rceil)) \rceil\!\}.$$

**Proof:** For any $s \in H(L, A)$ we have that $sq, sqq, sqqq, \ldots \in H(L, A)$, where $q \equiv \mathit{fs}(s)$, since $\models$ **null** $\Rightarrow \neg\lceil A \rceil$. Let $t \equiv sqqq\cdots$; then $\forall s \leq t(s \in H(L, A))$, hence $t \in \mathit{safe}(L, A)$, hence $t \in \{\!\lfloor \lceil S \rceil \rfloor\!\}$, and hence $s \in S$. This is true for any $s \in H(L, A)$, and so $H(L, A) \subseteq S$. ∎

**Theorem 8** *For any pair of action predicates $A$ and $A'$ s.t. $A' \subseteq A$, $C(A, A')$ is a domain, with least element $\bot = (Q^+, \emptyset)$; furthermore, the l.u.b. of any chain $\{Z_i\}$, where $Z_i = (S_i, L_i)$ for all $i$, is*

$$\bigsqcup_i Z_i = (\bigcap_i S_i, \bigcup_i L_i).$$

**Proof:** It is easily verified that $\bot$ is an element of $C(A, A')$, that $\bot \sqsubseteq Z'$ for all $Z' \in C(A, A')$, and that $\sqsubseteq$ is a partial order. Let $S \equiv \bigcap_i S_i$, $L \equiv \bigcup_i L_i$ and $Z \equiv (S, L)$, and assume that $Z \in C(A, A')$. Then $Z$ is trivially the l.u.b. of $\{Z_i\}$. It remains only to show that $Z = (S, L)$ is indeed an element of $C(A, A')$:

1. Since $L$ is the countable union of countable sets of $A'$-admissible transition rules, we have that $L$ is a countable set of $A'$-admissible transition rules.

2. For any $t \in \mathit{safe}(L, A)$ we have that $s \in H(L, A)$ for all $s \leq t$. In fact, for any (finite) $s \leq t$ we have that $s \in H(L', A)$ for some *finite* $L' \subseteq L$ since $\{ r \mid r \leq s \}$ is finite. Let $n_l$ for all $l \in L'$ be some $j$ s.t. $l \in L_j$, and let $n \equiv \max\{ n_l \mid l \in L' \}$; then, for all $i \geq n$, we see that $L' \subseteq L_n \subseteq L_i$, hence $s \in H(L_i, A)$, and by Property 7 we conclude that $s \in S_i$. Since $S_i \supseteq S_n$ for all $i \leq n$, we then have $s \in \bigcap_i S_i = S$. This holds for any $t \in \mathit{safe}(L, A)$ and $s \leq t$, so $\mathit{safe}(L, A) \subseteq \{\lfloor \lceil S \rceil \rfloor\}$.

Thus $(S, L) \in C(A, A')$. ∎

**Definition:** A ***core domain*** is any domain $\mathcal{D}$ which is an $i$-level core domain for some $i \in \mathbb{N}$, where

- $\mathcal{D}$ is a 0-level core-domain iff $\mathcal{D} = C(A, A')$ for some pair of action predicates $A$ and $A'$ s.t. $A' \subseteq A$;
- $\mathcal{D}$ is an $(i+1)$-level core-domain iff one of the following holds:
  - $\mathcal{D}$ is an $i$-level core domain;
  - $\mathcal{D} = \mathcal{D}_1^Y$ for some set $Y$ and $i$-level core domain $\mathcal{D}_1$;
  - $\mathcal{D} = (\mathcal{D}_1 \xrightarrow{c} \mathcal{D}_2)$ for some pair of $i$-level core domains $\mathcal{D}_1$ and $\mathcal{D}_2$;
  - $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ for some collection of $i$-level core domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$.

For each core domain there is a corresponding strong domain derived from $\wp(Q^\omega)$:

**Definition:** Given a core domain $\mathcal{D}$, the ***parallel domain*** of $\mathcal{D}$, denoted $pd(\mathcal{D})$, is

- $\wp(Q^\omega)$ if $\mathcal{D} = C(A, A')$ for some $A$ and $A'$;
- $pd(\mathcal{D}_1)^Y$ if $\mathcal{D} = \mathcal{D}_1^Y$ for some set $Y$ and core domain $\mathcal{D}_1$;
- $(pd(\mathcal{D}_1) \xrightarrow{m} pd(\mathcal{D}_2))$ if $\mathcal{D} = (\mathcal{D}_1 \xrightarrow{c} \mathcal{D}_2)$ for some pair of core domains $\mathcal{D}_1$ and $\mathcal{D}_2$;
- $pd(\mathcal{D}_1) \times \cdots \times pd(\mathcal{D}_n)$ if $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ for some collection of core domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$.

We now extend the notion of a core as given in Chapter 3.

**Definition:** Given a core domain $\mathcal{D}$ and $T \in pd(\mathcal{D})$, a *$\mathcal{D}$-core of $T$* is any $c \in \mathcal{D}$ s.t.

- if $\mathcal{D} = C(A, A')$ and $c = (S, L)$ for some $A$, $A'$, $S$ and $L$, then

$$T \supseteq \{\lfloor \lceil S \rceil \rfloor\} \cap \bigcap_{l \in L} \mathit{fair}(l);$$

- if $\mathcal{D} = \mathcal{D}_1^Y$ for some $Y$ and $\mathcal{D}_1$, then

$$\forall y \in Y (c(y) \text{ is a } \mathcal{D}_1\text{-core of } T(y));$$

- if $\mathcal{D} = (\mathcal{D}_1 \xrightarrow{c} \mathcal{D}_2)$ for some $\mathcal{D}_1$ and $\mathcal{D}_2$, then

$$\forall x \in \mathcal{D}_1, y \in pd(\mathcal{D}_1) (x \text{ is a } \mathcal{D}_1\text{-core of } y \Rightarrow c(x) \text{ is a } \mathcal{D}_2\text{-core of } T(y));$$

- if $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ for some $\mathcal{D}_1, \ldots, \mathcal{D}_n$, then
$$\forall i (c_i \text{ is a } \mathcal{D}_i\text{-core of } T_i),$$
where $c \equiv (c_1, \ldots, c_n)$ and $T \equiv (T_1, \ldots, T_n)$.

**Definition:** Given a core domain $\mathcal{D}$, we say that $T$ is $\mathcal{D}$*-founded* iff $T \in pd(\mathcal{D})$ and there exists a $\mathcal{D}$-core of $T$.

Note that for any $T \in \wp(Q^\omega)$ and action predicates $A$ and $A'$ s.t. $A' \subseteq A$, $T$ is $C(A, A')$-founded iff $T$ is $(A, A')$-founded.

### 5.2.3 Properties

We now prove a very important property of core domains.

**Lemma 9** *Given any core domain $\mathcal{D}$, $\bot_{\mathcal{D}}$ is a $\mathcal{D}$-core of $\bot_{pd(\mathcal{D})}$; furthermore, for any pair of chains $\{c_i\}$ in $\mathcal{D}$ and $\{T_i\}$ in $pd(\mathcal{D})$ s.t. $\bigsqcup_j c_j$ exists and $c_i$ is a $\mathcal{D}$-core of $T_i$ for all $i$, we have that $\bigsqcup_i c_i$ is a $\mathcal{D}$-core of $\bigsqcup_i T_i$.*

The second statement above is depicted by the following diagram, in which a thin vertical arrow from $c$ to $T$ means that $c$ is a $\mathcal{D}$-core of $T$, and a thick horizontal arrow from $x$ to $y$ means $x \sqsubseteq y$:

$$\begin{tikzcd}
T_0 \arrow[r] & T_1 \arrow[r] & T_2 \arrow[r] & T_3 \arrow[r] & {} & \cdots & \bigsqcup_i T_i \\
 & & & & & \Rightarrow & \\
c_0 \arrow[r] \arrow[uu] & c_1 \arrow[r] \arrow[uu] & c_2 \arrow[r] \arrow[uu] & c_3 \arrow[r] \arrow[uu] & {} & \cdots & \bigsqcup_i c_i \arrow[uu]
\end{tikzcd}$$

**Proof:** By induction on the structure of $\mathcal{D}$.

Base step: For any pair of action predicates $A$ and $A'$ s.t. $A' \subseteq A$ it is easily seen that $\bot_{C(A,A')} = (Q^+, \emptyset)$ is a $C(A, A')$-core of $\bot_{pd(C(A,A'))} = Q^\omega$. Given any pair of chains $\{c_i\}$ in $C(A, A')$ and $\{T_i\}$ in $\wp(Q^\omega)$ satisfying the above conditions, and letting $c_i = (S_i, L_i)$ for all $i$ we have that

$$\bigsqcup_i T_i = \bigcap_i T_i \supseteq \bigcap_i (\{\lfloor \lceil S_i \rceil \rfloor\} \cap \bigcap_{l \in L_i} fair(l)) = \{\lfloor \lceil S \rceil \rfloor\} \cap \bigcap_{l \in L} fair(l)$$

where $S \equiv \bigcap_i S_i$ and $L \equiv \bigcup_i L_i$. Since $(S, L) = \bigsqcup_i c_i$, we then have that $\bigsqcup_i c_i$ is a $C(A, A')$-core of $\bigsqcup_i T_i$.

Induction step: We give the proof only for the case that $\mathcal{D} = (\mathcal{D}_1 \xrightarrow{c} \mathcal{D}_2)$ for some pair of core domains $\mathcal{D}_1$ and $\mathcal{D}_2$. The proof for the other two cases is similar.

By the induction hypothesis, for all $x \in \mathcal{D}_1$ and $y \in pd(\mathcal{D}_1)$, $\bot_{\mathcal{D}}(x) = \bot_{\mathcal{D}_2}$ is a $\mathcal{D}_2$-core of $\bot_{pd(\mathcal{D})}(x) = \bot_{pd(\mathcal{D}_2)}$; hence $\bot_{\mathcal{D}}$ is a $\mathcal{D}$-core of $\bot_{pd(\mathcal{D})}$. For the second part, we have for any $x \in \mathcal{D}_1$ and $y \in pd(\mathcal{D}_1)$ that

$$(\forall i : x \text{ is a } \mathcal{D}_1\text{-core of } y \Rightarrow c_i(x) \text{ is a } \mathcal{D}_2\text{-core of } T_i(y))$$

(since $c_i$ is a $\mathcal{D}$-core of $T_i$ for all $i$) and hence, using the induction hypothesis,

$$\begin{aligned}
x \text{ is a } \mathcal{D}_1\text{-core of } y &\Rightarrow (\forall i : c_i(x) \text{ is a } \mathcal{D}_2\text{-core of } T_i(y)) \\
&\Rightarrow \textstyle\bigsqcup_i c_i(x) \text{ is a } \mathcal{D}_2\text{-core of } \bigsqcup_i T_i(y) \\
&\Rightarrow (\textstyle\bigsqcup_i c_i)(x) \text{ is a } \mathcal{D}_2\text{-core of } (\bigsqcup_i T_i)(y).
\end{aligned}$$

So $\bigsqcup_i c_i$ is a $(\mathcal{D}_1 \xrightarrow{c} \mathcal{D}_2)$-core of $\bigsqcup_i T_i$. ∎

As a result of the above, we have the following result:

**Theorem 10** *Given any core domain $\mathcal{D}$, if $F$ is $(\mathcal{D} \xrightarrow{c} \mathcal{D})$-founded, then $\mathit{fix}(F)$ is $\mathcal{D}$-founded.*

**Proof:** Let $c$ be a $(\mathcal{D} \xrightarrow{c} \mathcal{D})$-core of $F$. Furthermore, let $\gamma_0 \equiv \bot_{\mathcal{D}}$ and $\gamma'_0 \equiv \bot_{pd(\mathcal{D})}$, $\gamma_{i+1} \equiv c(\gamma_i)$ and $\gamma'_{i+1} \equiv F(\gamma'_i)$ for any ordinal $i$, and $\gamma_i \equiv \bigsqcup_{j<i} \gamma_j$ and $\gamma'_i \equiv \bigsqcup_{j<i} \gamma'_j$ for any limit ordinal $i$. Due to the continuity of $c$, $\gamma_i$ is well-defined and equal to $\gamma_\omega$ for all ordinals $i > \omega$.

We show by transfinite induction that $\gamma_i$ is a $\mathcal{D}$-core of $\gamma'_i$ for all ordinals $i$. Base step: By the previous theorem, $\gamma_0$ is a $\mathcal{D}$-core of $\gamma'_0$. Induction step: if $\gamma_i$ is a $\mathcal{D}$-core of $\gamma'_i$, then since $c$ is a $(\mathcal{D} \xrightarrow{c} \mathcal{D})$-core of $F$ we have that $c(\gamma_i) = \gamma_{i+1}$ is a $\mathcal{D}$-core of $F(\gamma'_i) = \gamma'_{i+1}$. Furthermore, for any limit ordinal $i$, $\bigsqcup_{j<i} \gamma_j$ exists and hence by the previous theorem we have that if $\gamma_j$ is a $\mathcal{D}$-core of $\gamma'_j$ for all $j < i$, then $\gamma_i = \bigsqcup_{j<i} \gamma_j$ is a $\mathcal{D}$-core of $\gamma'_i = \bigsqcup_{j<i} \gamma'_j$.

Since we have that $\mathit{fix}(F) = \gamma'_i$ for some ordinal $i$, we then have that $\gamma_i$ is a $\mathcal{D}$-core of $\mathit{fix}(F)$. ∎

The definitions and results of this section are presented in a form more general than needed for our immediate purposes, so that they may be applied to programming languages other than those examined in this thesis.

## 5.3 The language $SP$

We now turn our attention to the languages $SP$ used to define sequential processes in the previous chapter. We have already established that $\mathcal{S}$ is well-defined. The question we now address is this: given some $P \in SP$ and action predicates $A$ and $A'$ s.t. $A' \subseteq A$, how can we guarantee that $\mathcal{M}[\![P]\!]$ is $(A, A')$-founded?

In order to do this, we need the following theorem.

**Theorem 11** *If*

1. *$A$ and $A'$ are action predicates s.t. $A' \subseteq A$,*
2. *$a$ and $a'$ are state relations s.t. $A = \{\!\lceil a \rceil\!\}$ and $A' = \{\!\lceil a' \rceil\!\}$, and*
3. *for $1 \le i \le n$, $l_i = (\pi_i, \tau_i)$ is an $A'$-admissible transition rule,*

*then the function*

$$\mathit{cmd}(l_1, \ldots, l_n) : \wp(Q^\omega)^n \to \wp(Q^\omega)$$

*(which we abbreviate as $f$) defined by*

$$f(T_1, \ldots, T_n) \equiv \{\!\lfloor \bigvee_{i=1}^{n} \Diamond(\ominus(\boxminus \neg a \wedge \lceil \pi_i \rceil) \wedge \lceil \tau_i \rceil \wedge \lfloor T_i \rfloor) \vee (\Box \neg a \wedge \Diamond \boxplus \neg \bigvee_{i=1}^{n} \lceil \pi_i \rceil) \rfloor\!\}$$

*for all $T_1, \ldots, T_n \in Q^\omega$, is $(C(A, A')^n \xrightarrow{c} C(A, A'))$-founded.*

The proof of the above theorem is rather long and tedious, and may be found in the appendix. In the sequel we will write

$$\mathit{cmd}[\![e_1 \to c_1 \mid \cdots \mid e_n \to c_n]\!]$$

for

$$\mathit{cmd}((\{\!\lceil e_1 \rceil\!\}, \{\!\lceil c_1 \rceil\!\}), \ldots, (\{\!\lceil e_n \rceil\!\}, \{\!\lceil c_n \rceil\!\})).$$

With this we can now prove the result we need:

**Theorem 12** *For any pair of state relations $a$ and $a'$, and $R \in BD$ s.t.*

1. *$\{\lceil a \rceil\}$ and $\{\lceil a' \rceil\}$ are action predicates and $\models a' \Rightarrow a$, and*

2. *for every command in $R$ of form $[e \xrightarrow{\mathsf{j}} c]$ or $[e \xrightarrow{\mathsf{f}} c]$ there exists some state relation $c'$ s.t. $\models c' \Rightarrow c$ and $(\{\lceil e \rceil\}, \{\lceil c' \rceil\})$ is $\{\lceil a' \rceil\}$-admissible,*

*the trace set $\mathcal{M}[\![\alpha : a : R]\!]$ is $(\{\lceil a \rceil\}, \{\lceil a' \rceil\})$-admissible.*

**Proof:** Let $A \equiv \{\lceil a \rceil\}$ and $A' \equiv \{\lceil a' \rceil\}$. Examining the definition of $\mathcal{M}$, we see that $\mathcal{M}[\![\alpha : a : R]\!]$ is $(A, A')$-founded if $\mathcal{S}_a[\![R]\!]$ is $(A, A')$-founded. Let $BD'$ be the set of elements of $BD$ satisfying condition 2. above; then we see that for any $R' \in BD'$, in the recursive definition of $\mathcal{S}_a[\![R']\!]$ the function $\mathcal{S}_a$ is applied only to elements of $BD'$, and so we can restrict $\mathcal{S}_a$ to operate only on elements of $BD'$ and still use this recursive definition. $\mathcal{S}_a[\![R]\!]$ is $(A, A')$-founded if $\mathcal{S}_a$ thus restricted is $\mathcal{D}$-founded, where

$$\mathcal{D} \equiv C(A, A')^{BD'},$$

and by by Theorem 10 this will be so if the function $F \colon \mathcal{D}' \xrightarrow{\mathrm{m}} \mathcal{D}'$ is $(\mathcal{D} \xrightarrow{\mathrm{c}} \mathcal{D})$-founded, where

$$\mathcal{D}' \equiv pd(D) \qquad (= \wp(Q^{\omega})^{BD'})$$

and $F$ is the function (implicitly given by the recursive definition of $\mathcal{S}_a$) s.t. $\mathcal{S}_a = \mathit{fix}(F)$.

For all $f \in \mathcal{D}'$, we define

- $F'(f)[\![\mathbf{end}]\!] \equiv \{\lfloor \neg a \rfloor\}$;
- $F'(f)[\![\mathbf{skip}; R]\!] \equiv f[\![R]\!]$;
- $F'(f)[\![\mathbf{abort}; R]\!] \equiv \{\lfloor \mathbf{tt} \rfloor\}$;
- $F'(f)[\![[e \xrightarrow{\mathsf{j}} c]; R]\!] \equiv \mathit{cmd}[\![e \rightarrow c']\!](f[\![R]\!])$, where $c'$ is some state relation satisfying condition 2. above;
- $F'(f)[\![[e \xrightarrow{\mathsf{f}} c]; R]\!] \equiv \mathit{cmd}[\![e \rightarrow c']\!](f[\![R]\!])$, where $c'$ is some state relation satisfying condition 2. above;
- $F'(f)[\![[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R']; R]\!] \equiv f[\![R''; R]\!]$, where $R''$ is obtained from $R'$ by simultaneously replacing each free instance of any $p_i$ in $R'$ by '$[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R_i]$';
- $F'(f)[\![p; R]\!] \equiv \{\lfloor \mathbf{tt} \rfloor\}$ if $p \in PN$;
- $F'(f)[\![[e_1 \rightarrow R_1 \mid \cdots \mid e_n \rightarrow R_n]; R]\!] \equiv$
  $\mathit{cmd}[\![e_1 \rightarrow \mathbf{null} \mid \cdots \mid e_n \rightarrow \mathbf{null}]\!](f[\![R_1; R]\!], \ldots, f[\![R_n; R]\!])$.

Using the facts that, for any state formula $e$,

1. $\models \Diamond \boxplus \neg e \Rightarrow \Box \diamondplus \neg e$ and

2. $\models \ominus(e \wedge \boxminus \neg a) \wedge \mathbf{null} \Rightarrow e \wedge \boxminus \neg a$,

it is straightforward to verify that $F \sqsubseteq F'$. Then, since

$$T \supseteq T' \wedge T' \text{ is } C(A, A')\text{-founded} \Rightarrow T \text{ is } C(A, A')\text{-founded}$$

for any $T, T' \subseteq Q^{\omega}$, if $F'$ is $(\mathcal{D} \xrightarrow{c} \mathcal{D})$-founded then so is $F$.

For any $e_1, \ldots e_n \in EX_1$ and $c_1, \ldots c_n \in EX_2$, let

$$cmdc[\![e_1 \rightarrow c_1 \mid \cdots \mid e_n \rightarrow c_n]\!]$$

be some $(C(A, A')^n \xrightarrow{c} C(A, A'))$-core of

$$cmd[\![e_1 \rightarrow c_1 \mid \cdots \mid e_n \rightarrow c_n]\!]$$

if such exists. We define $G: \mathcal{D} \rightarrow \mathcal{D}$ by

- $G(g)[\![\mathbf{end}]\!] \equiv (\{\lceil \neg a \rceil\}, \emptyset)$;
- $G(g)[\![\mathbf{skip}; R]\!] \equiv g[\![R]\!]$;
- $G(g)[\![\mathbf{abort}; R]\!] \equiv (\{\lceil \mathbf{tt} \rceil\}, \emptyset)$;
- $G(g)[\![[e \xrightarrow{j} c]; R]\!] \equiv cmdc[\![e \rightarrow c']\!](g[\![R]\!])$, where $c'$ is some state relation satisfying condition 2. above;
- $G(g)[\![[e \xrightarrow{f} c]; R]\!] \equiv cmdc[\![e \rightarrow c']\!](g[\![R]\!])$, where $c'$ is some state relation satisfying condition 2. above;
- $G(g)[\![[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R']; R]\!] \equiv g[\![R''; R]\!]$, where $R''$ is obtained from $R'$ by simultaneously replacing each free instance of any $p_i$ in $R'$ by '$[p_1 = R_1 \mid \cdots \mid p_n = R_n \mid R_i]$';
- $G(g)[\![p; R]\!] \equiv (\{\lceil \mathbf{tt} \rceil\}, \emptyset)$ if $p \in PN$;
- $G(g)[\![[e_1 \rightarrow R_1 \mid \cdots \mid e_n \rightarrow R_n]; R]\!] \equiv$
  $cmdc[\![e_1 \rightarrow \mathbf{null} \mid \cdots \mid e_n \rightarrow \mathbf{null}]\!](g[\![R_1; R]\!], \ldots, g[\![R_n; R]\!])$,

for all $g: BD' \rightarrow C(A, A')$. Noting that $(\{\lceil e \rceil\}, \{\lceil \mathbf{null} \rceil\})$ is $A'$-admissible for any local formula $e$, we see that Theorem 11 applies and hence the appropriate core exists for every use of *cmdc* in the above definition. From this it is straightforward to verify that $G$ is a $(\mathcal{D} \xrightarrow{c} \mathcal{D})$-core of $F'$, and hence $F'$ is $(\mathcal{D} \xrightarrow{c} \mathcal{D})$-founded. ∎

# Chapter 6

# Conclusion

## 6.1 What have we accomplished?

First of all, we have shown how the behavior of a concurrent system may be described as the conjunction of the behaviors of its component agents and a global constraint on the system behavior. We presented a notation for defining the behavior of some simple agents, and used it to give the semantics of digital circuits and Petri nets.

Secondly, we have shown how to give the semantics of a more expressive class of programming notations: those programming languages which describe the concurrent operation of a set of sequential processes, each process described by composing a set of primitive commands using sequential composition, IF statements, and recursion. This class includes languages which use shared variables and P and V operations, as well as CSP. A specific contribution of this thesis has been to give a formal semantics for CSP augmented with the probe.

Thirdly, perhaps the major contribution of this thesis is the method we have presented for showing that the meaning of a program is always a nonempty trace set, and that therefore it may serve as the basis for a consistent axiom set in an axiomatic semantics. Furthermore, the proof of this for each programming language we examined was not overly specific to the particular language, but instead made use of some fairly general theorems.

In addition, the technique of core domains (Chapter 5) may be of interest to other researchers in program semantics who are faced with the problem that the objects of interest are a subset $Y$ of a domain $\mathcal{D}$, but do not form a subdomain, although there is reason to believe that the least fixed-point of any 'reasonable' function on the domain will belong to this subset.

## 6.2 Directions for further research

An obvious next step is to devise a proof system for the two languages examined in Chapter 4, based on the semantics we have given. Such a proof system would of course use some form of temporal logic.

One complaint that could be made about the semantics given for the two languages in Chapter 4 is that they are insufficiently abstract, since the values of variables internal to a process are made visible as part of the system state. This could be remedied in two ways. The first way is to redefine $\mathcal{S}_a$ so that it has functionality

$$\mathcal{S}_a : BD \to \wp(Q^\omega)^I,$$

where $I$ is the set of possible internal states of a process. $\mathcal{S}_a$ would then give the future behavior of a process given its internal state and the remainder of the program it is to execute. The second way is to introduce a 'hiding' operator which maps traces in $Q^\omega$ to traces in $Q'^\omega$, where the states $q' \in Q'$ are the restrictions of the states $q \in Q$ to the set of externally observable quantities. Such an operator would be the analog of the projection operator of trace theory [23].

A hiding operator would also be of use in giving the semantics of a language in which recursion could be used at the level of process composition. In order to show that the meaning of any program in such a language is nonempty, we would need a proof that the intersection operation is $\mathcal{D}$-founded for the appropriate core domain $\mathcal{D}$, and similarly with the hiding operation. The author already has such a proof for the intersection operation, based on a modification of Theorem 4, but not for the hiding operation.

# Appendix A

# Proof of Theorem 11

The proof of the following theorem is long and tedious. We suggest that you get a good night's sleep before reading it.

**Theorem 11** *If*

1. *$A$ and $A'$ are action predicates s.t. $A' \subseteq A$,*
2. *$a$ and $a'$ are state relations s.t. $A = \{\lceil a \rceil\}$ and $A' = \{\lceil a' \rceil\}$, and*
3. *for $1 \le i \le n$, $l_i = (\pi_i, \tau_i)$ is an $A'$-admissible transition rule,*

*then the function*

$$cmd(l_1, \ldots, l_n) \colon (Q^\omega)^n \to Q^\omega$$

*(which we abbreviate as $f$) defined by*

$$f(T_1, \ldots, T_n) \equiv \{\lfloor \bigvee_{i=1}^{n} \Diamond(\ominus(\boxminus\neg a \wedge \lceil \pi_i \rceil) \wedge \lceil \tau_i \rceil \wedge \lfloor T_i \rfloor) \vee (\Box\neg a \wedge \Diamond\boxplus\neg \bigvee_{i=1}^{n} \lceil \pi_i \rceil)\rfloor\}$$

*for all $T_1, \ldots, T_n \in Q^\omega$, is $(C(A, A')^n \xrightarrow{c} C(A, A'))$-founded.*

**Proof:** $f$ is obviously a monotonic function. Thus we need only to construct a $(C(A, A')^n \xrightarrow{c} C(A, A'))$-core for $f$.

For $1 \le i \le n$, let

$$\begin{aligned}
\rho_i &\equiv \{\lceil \lceil \tau_i \rceil \wedge \ominus(\lceil \pi_i \rceil \wedge \boxminus\neg a)\rceil\} \\
\rho &\equiv \bigcup_{j=1}^{n} \rho_j \\
\rho'_i &\equiv \{\lceil \lceil \rho_i \rceil \wedge \neg \ominus \diamondminus \lceil \rho \rceil \wedge \bigwedge_{j<i} \neg \lceil \rho_j \rceil \rceil\} \\
\rho' &\equiv \bigcup_{j=1}^{n} \rho_j
\end{aligned}$$

Note that

1. $\models \Box\neg\lceil \rho \rceil \vee \bigvee_{i=1}^{n} \Diamond\lceil \rho'_i \rceil$,

2. $\models \lceil \rho'_i \rceil \Rightarrow \bigwedge_{j \neq i} \Box \neg \lceil \rho'_j \rceil$ for all $i$.

For any $v \in Q^\infty$ there is at most one choice of $s$, $w$ and $i$ s.t. $s \in \{\lceil \rho'_i \rceil\}$, $fs(s) = is(w)$ and $v = s \cdot w$; let $\beta(v) \equiv s$ and $\alpha(v) \equiv w$ whenever such exist.

For $Z_i = (S_i, L_i) \in C(A, A')$ $(1 \leq i \leq n)$ we define

$$g(Z_1, \ldots, Z_n) \equiv (S, L)$$

where

$$\begin{aligned} S &\equiv \{\lceil \boxminus(\neg a \wedge \neg \lceil \rho \rceil) \rceil\} \cup \bigcup_{i=1}^{n} \rho'_i \cdot S_i \\ L &\equiv \{(\pi_0, \rho)\} \cup \bigcup_{i=1}^{n} \{ (\rho'_i \cdot \pi, \rho'_i \cdot \tau) \mid (\pi, \tau) \in L_i \} \\ \pi_0 &\equiv \{\lceil \bigvee_{i=1}^{n} \lceil \pi_i \rceil \wedge \boxminus(\neg a \wedge \neg \lceil \rho \rceil) \rceil\} \end{aligned}$$

(recall that, in general, $V \cdot W \equiv \{ v \cdot w \mid fs(v) = is(w) \wedge v \in V \wedge w \in W \}$ if $V \subseteq Q^+$).

First we note that for any $t \in \{\lfloor \lceil S \rceil \rfloor\}$ and for any $i$,

$$\begin{aligned} t \models \Box \neg \lceil \rho \rceil &\Rightarrow t \models \boxminus(\neg a \wedge \neg \lceil \rho \rceil) \\ &\Rightarrow t \in \{\lfloor \neg a \wedge \neg \lceil \rho \rceil \rfloor\} \\ t \models \Diamond \lceil \rho'_i \rceil &\Rightarrow t \models \bigwedge_{j \neq i} \neg \lceil \rho'_j \rceil \\ &\Rightarrow t \models (\diamondminus \lceil \rho'_i \rceil \Rightarrow \lceil \rho'_i \cdot S_i \rceil) \\ &\Rightarrow \alpha(t) \models \lceil S_i \rceil \\ &\Rightarrow t \in \rho'_i \cdot \{\lfloor \lceil S_i \rceil \rfloor\} \end{aligned}$$

and hence

$$\{\lfloor \lceil S \rceil \rfloor\} \subseteq \{\lfloor \neg a \wedge \neg \lceil \rho \rceil \rfloor\} \cup \bigcup_{i=1}^{n} \rho'_i \cdot \{\lfloor \lceil S_i \rceil \rfloor\}.$$

Since the reverse inclusion is easily seen to hold, we then have that

$$\{\lfloor \lceil S \rceil \rfloor\} = \{\lfloor \neg a \wedge \neg \lceil \rho \rceil \rfloor\} \cup \bigcup_{i=1}^{n} \rho'_i \cdot \{\lfloor \lceil S_i \rceil \rfloor\}.$$

Second, we show that $g(Z_1, \ldots, Z_n) \in C(A, A')$:

1. Since each $(\pi_i, \tau_i)$ is $A'$-admissible, so is $(\pi_0, \rho)$. For any $i$ and $(\pi, \tau) \in L_i$, $(\pi, \tau)$ is $A'$-admissible and hence

   (a) for all $s \in \rho'_i \cdot \pi$, letting $r \equiv \beta(s)$ and $s' \equiv \alpha(s)$, we have that

$$\begin{aligned} s \in \rho'_i \cdot \pi &\Rightarrow s' \in \pi \\ &\Rightarrow \exists q (s'q \in \tau) \\ &\Rightarrow \exists q (r \cdot s'q \in \rho'_i \cdot \tau) \\ &\Rightarrow \exists q (sq \in \rho'_i \cdot \tau); \end{aligned}$$

(b) since $A' = \{\lceil a' \rceil\}$, where $a'$ is a state relation, we have that $s \in A' \Rightarrow r \cdot s \in A'$ for all $r, s \in Q^+$, hence $\rho_i' \cdot A' \subseteq A'$, and so

$$\begin{aligned}\{\, sq \mid s \in \rho_i' \cdot \pi \wedge sq \in \rho_i' \cdot \tau \,\} &= \{\, r \cdot s'q \mid r \in \rho_i' \wedge \mathit{fs}(r) = \mathit{is}(s') \wedge s' \in \pi \wedge s'q \in \tau \,\} \\ &= \rho_i' \cdot \{\, s'q \mid s' \in \pi \wedge s'q \in \tau \,\} \\ &\subseteq \rho_i' \cdot (A' \cup \{\lceil \mathbf{null} \rceil\}) \\ &\subseteq A' \cup \{\lceil \mathbf{null} \rceil\}.\end{aligned}$$

Thus $(\rho_i' \cdot \pi, \rho_i' \cdot \tau)$ is $A'$-admissible for all $i$ and $(\pi, \tau) \in L_i$, and so $L$ is a countable set of $A'$-admissible transition rules.

2. For any $t \in \mathit{safe}(L, A)$, using the fact that

$$\models \lceil \rho \rceil \wedge \ominus \lceil \pi_0 \rceil \Rightarrow \lceil \rho' \rceil,$$

we have that

$$t \models \neg a \vee \lceil \rho' \rceil \vee \exists i \exists (\pi, \tau) \in L_i (\lceil \rho_i' \cdot \tau \rceil \wedge \ominus \lceil \rho_i' \cdot \pi \rceil)$$

and so

$$\begin{aligned}t \models \Box \neg \lceil \rho \rceil &\Rightarrow t \models \neg a \wedge \neg \rho \\ &\Rightarrow t \in \{\lfloor \lceil S \rceil \rfloor\} \\ t \models \Diamond \lceil \rho_i' \rceil &\Rightarrow t \models \bigwedge_{j \neq i} \Box \neg \lceil \rho_j' \rceil \\ &\Rightarrow t \models (\Diamond \lceil \rho_i' \rceil \Rightarrow \neg a \vee \lceil \rho_i' \rceil \vee \exists (\pi, \tau) \in L_i (\lceil \rho_i' \cdot \tau \rceil \wedge \ominus \lceil \rho_i' \cdot \pi \rceil)) \\ &\Rightarrow \alpha(t) \models \neg a \vee \mathbf{null} \vee \exists (\pi, \tau) \in L_i (\lceil \tau \rceil \wedge \ominus \lceil \pi \rceil) \\ &\Rightarrow \alpha(t) \in \mathit{safe}(L_i, A) \subseteq \{\lfloor \lceil S_i \rceil \rfloor\} \\ &\Rightarrow t \in \rho_i' \cdot \{\lfloor \lceil S_i \rceil \rfloor\} \subseteq \{\lfloor \lceil S \rceil \rfloor\}.\end{aligned}$$

Thus $\mathit{safe}(L, A) \subseteq \{\lfloor \lceil S \rceil \rfloor\}$.

Third, we show that, for any $T_1, \ldots, T_n \subseteq Q^\omega$ s.t. $(S_i, L_i)$ is a $C(A, A')$-core of $T_i$ for all $i$, $(S, L)$ is a $C(A, A')$-core of $f(T_1, \ldots, T_n)$. For any $t \in \{\lfloor \lceil S \rceil \rfloor\} \cap \bigcap_{l \in L} \mathit{fair}(l)$, there are two cases:

1. If $t \models \Box \neg \lceil \rho \rceil$, then $t \in f(T_1, \ldots, T_n)$, since

$$\begin{aligned}&\{\lfloor \lceil S \rceil \rfloor\} \cap \{\lfloor \Box \neg \lceil \rho \rceil \rfloor\} \cap \bigcap_{l \in L} \mathit{fair}(l) \\ \subseteq\;& \{\lfloor \neg a \wedge \neg \lceil \rho \rceil \rfloor\} \cap \mathit{fair}(\pi_0, \rho) \\ =\;& \{\lfloor \Box(\neg a \wedge \neg \lceil \rho \rceil) \wedge \neg \Box \oplus \lceil \pi_0 \rceil \rfloor\} \\ =\;& \{\lfloor \Box(\neg a \wedge \neg \lceil \rho \rceil) \wedge \neg \Box \oplus \bigvee_{i=1}^{n} \lceil \pi_i \rceil \rfloor\} \\ & (\text{using the fact that } \models \Box(\neg a \wedge \neg \lceil \rho \rceil) \Rightarrow (\lceil \pi_0 \rceil \Leftrightarrow \textstyle\bigvee_{i=1}^{n} \lceil \pi_i \rceil)) \\ \subseteq\;& \{\lfloor \Box \neg a \wedge \Diamond \boxplus \neg \bigvee_{i=1}^{n} \lceil \pi_i \rceil \rfloor\} \\ \subseteq\;& f(T_1, \ldots, T_n)\end{aligned}$$

2. If $t \models \Diamond\lceil\rho_i'\rceil$ for some $i$, then $t \in \rho_i' \cdot \{\!\lfloor \lceil S_i \rceil \rfloor\!\}$ and hence $\alpha(t) \in \{\!\lfloor \lceil S_i \rceil \rfloor\!\}$. In addition, letting $r \equiv \beta(t)$, for all $(\pi, \tau) \in L_i$ we have

$$\begin{aligned}
\alpha(t) \models \Box\Diamond\!\!\!\!+\lceil\pi\rceil &\Rightarrow \{\, s \le \alpha(t) \mid s \in \pi \,\} \text{ is infinite} \\
&\Rightarrow \{\, s' \le t \mid s' \in \rho_i' \cdot \pi \,\} \text{ is infinite} \\
&\Rightarrow \{\, s'q \le t \mid s' \in \rho_i' \cdot \pi \wedge s'q \in \rho_i' \cdot \tau \,\} \text{ is infinite} \\
&\phantom{\Rightarrow}\ \text{(using the fact that } t \in \mathit{fair}(\rho_i' \cdot \pi, \rho_i' \cdot \tau)) \\
&\Rightarrow \{\, r \cdot sq \mid r \cdot sq \le t \wedge s \in \pi \wedge sq \in \tau \,\} \text{ is infinite} \\
&\Rightarrow \{\, sq \le \alpha(t) \mid s \in \pi \wedge sq \in \tau \,\} \text{ is infinite} \\
&\Rightarrow \alpha(t) \models \Box\Diamond\!\!\!\!+(\lceil\tau\rceil \wedge \ominus\lceil\pi\rceil)
\end{aligned}$$

and hence $\alpha(t) \in \mathit{fair}(\pi, \tau)$. Since $(S_i, L_i)$ is a $C(A, A')$-core of $T_i$, this means that $\alpha(t) \in T_i$, and hence $t \in \rho_i' \cdot T_i$. Then $t \in f(T_1, \ldots, T_n)$, since

$$\begin{aligned}
t \in \rho_i' \cdot T_i &\Rightarrow t \in \{\!\lfloor \Diamond(\lceil\rho_i'\rceil \wedge \lfloor T_i \rfloor) \rfloor\!\} \\
&\Rightarrow t \in \{\!\lfloor \Diamond(\ominus(\boxminus\neg a \wedge \lceil\pi_i\rceil) \wedge \lceil\tau_i\rceil \wedge \lfloor T_i \rfloor) \rfloor\!\} \\
&\Rightarrow t \in f(T_1, \ldots, T_n)
\end{aligned}$$

Thus $\{\!\lfloor \lceil S \rceil \rfloor\!\} \cap \bigcap_{l \in L} \mathit{fair}(l) \subseteq f(T_1, \ldots, T_n)$.

Finally, we must show that $g$ is continuous. It is easily seen that it is monotonic by examining the definitions. We will show that it is continuous in each argument.

Let $1 \le i \le n$ and let $\{\, Z_{i,k} \mid k \in \mathsf{N} \,\}$ be a chain in $C(A, A')$, where $Z_{i,k} = (S_{i,k}, L_{i,k})$ for all $k$. First, note that $\bigcap_k(\rho_i' \cdot S_{i,k}) = \rho_i' \cdot (\bigcap_k S_{i,k})$. Proof: We show that the two sets are subsets of each other. From the monotonicity of the $\cdot$ operator on $\wp(Q^+)$ we have immediately that $\rho_i' \cdot (\bigcap_k S_{i,k}) \subseteq \bigcap_k(\rho_i' \cdot S_{i,k})$. The reverse inclusion also holds, since for all $s$,

$$\begin{aligned}
s \in \bigcap_k(\rho_i' \cdot S_{i,k}) &\Rightarrow \forall k(\exists r \le s(r \in \rho_i') \wedge \alpha(s) \in S_{i,k}) \\
&\Rightarrow \exists r \le s(r \in \rho_i') \wedge \forall k(\alpha(s) \in S_{i,k}) \\
&\Rightarrow s \in \rho_i' \cdot (\bigcap_k S_{i,k}).
\end{aligned}$$

Letting $(S, L) \equiv g(Z_1, \ldots, \bigsqcup_k Z_{i,k}, \ldots, Z_n)$, $(S', L') \equiv \bigsqcup_k g(Z_1, \ldots, Z_{i,k}, \ldots, Z_n)$ and $Z_j = (S_j, L_j)$ for all $j$, we show that $(S, L) = (S', L')$, as follows:

Let $V \equiv \{\!\lfloor \boxminus(\neg a \wedge \neg\lceil\rho\rceil) \rfloor\!\} \cup \bigcup_{j \ne i} \rho_j' \cdot S_j$; then

$$\begin{aligned}
S &= V \cup \rho_i' \cdot (\bigcap_k S_{i,k}) \\
&= V \cup \bigcap_k \rho_i' \cdot S_{i,k} \\
&= \bigcap_k (V \cup \rho_i' \cdot S_{i,k}) \\
&= S'.
\end{aligned}$$

Let $M \equiv \{(\pi_0, \rho)\} \cup \bigcup_{j \ne i}\{\, (\rho_i' \cdot \pi, \rho_i' \cdot \tau) \mid (\pi, \tau) \in L_j \,\}$; then

$$L = M \cup \{\, (\rho_i' \cdot \pi, \rho_i' \cdot \tau) \mid (\pi, \tau) \in \bigcup_k L_{i,k} \,\}$$

$$\begin{aligned} &= M \cup \bigcup_k \{ (\rho'_i \cdot \pi, \rho'_i \cdot \tau) \mid (\pi, \tau) \in L_{i,k} \} \\ &= \bigcup_k (M \cup \{ (\rho'_i \cdot \pi, \rho'_i \cdot \tau) \mid (\pi, \tau) \in L_{i,k} \}) \\ &= L'. \end{aligned}$$

Thus we have shown that $g$ is a $(C(A, A')^n \xrightarrow{c} C(A, A'))$-core of $f$. ∎

# Bibliography

[1] J. W. de Bakker and J. I. Zucker. "Processes and the Denotational Semantics of Concurrency". *Information and Control* 54 (1982), pp. 70–120.

[2] H. Barringer, et al. "Now You May Compose Temporal Logic Specifications". *16th ACM Symposium on the Theory of Computing*, (May, 1984).

[3] H. Barringer, et al. "A Really Abstract Concurrent Model and its Temporal Logic". *13th ACM Symposium on Principles of Programming Languages*, (Jan., 1986).

[4] H. Barringer and R. Kuiper. "Hierarchical Development of Concurrent Systems in a Temporal Logic Framework". *LNCS 197: Seminar on Concurrency*. Springer-Verlag, 1984.

[5] D. L. Black. *On the Existence of Delay-Insensitive Fair Arbiters: Trace Theory and its Limitations.* Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, tech. report CMU-CS-85-173, Oct. 1985.

[6] M. Chandy and J. Misra. *An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection.* Dept. of Computer Science, The University of Texas at Austin, tech. report TR-85-26, Nov. 1985.

[7] S. D. Brookes, et al. "A Theory of Communicating Sequential Processes". *Journal of the ACM* 31, no. 3 (July 1984), pp. 560–599.

[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[9] E. W. Dijkstra. "Cooperating Sequential Processes". *Programming Languages and Systems* (F. Genvys, ed.), pp. 43–112. Academic Press, New York, 1968.

[10] S. Eisenberg. *Automata, Languages and Machines* (vol. A), pp. 358–359. Academic Press, New York, 1974.

[11] N. Francez, et al. "A Linear-History Semantics for Languages for Distributed Programming". *Theoretical Computer Science* 32 (1984), pp. 25–46.

[12] C. A. R. Hoare. "Communicating Sequential Processes". *Communications of the ACM* 21, no. 8 (Aug. 1978), pp. 666–677.

[13] L. Lamport. "An Axiomatic Semantics of Concurrent Programming Languages". *Logics and Models of Concurrent Systems* (K. R. Apt, ed.), pp. 77–122. Springer-Verlag, New York, 1985.

[14] D. Lehmann, et al. "Impartiality, justice and fairness: the ethics of concurrent termination". *LNCS 115: Proc. 8th ICALP* (O. Kariv, S. Even, eds.). Springer-Verlag, New York, 1985.

[15] O. Lichtenstein and A. Pnueli. "The Glory of the Past". *LNCS 193: Logics of Programs.* Springer-Verlag, New York, 1985.

[16] Z. Manna. *Mathematical Theory of Computation*, chapter 5. McGraw-Hill, New York, 1974.

[17] Z. Manna and A. Pnueli. "Verification of Concurrent Programs: the Temporal Framework". *The Correctness Problem in Computer Science* (R. S. Boyer and J. S. Moore, eds.), pp. 215–273. Academic Press, New York, 1982.

[18] A. J. Martin. "The Probe: An Addition to Communication Primitives". *Information Processing Letters* 20, no. 1 (Jan. 1985), pp. 125–130.

[19] R. Milner. *Lecture Notes in Computer Science, vol. 92: A Calculus of Communicating Systems.* Springer-Verlag, New York, 1980.

[20] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*, pp. 6–8. North-Holland, Amsterdam, 1974.

[21] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[22] M. Rem. "Concurrent Computations and VLSI Circuits". *Control Flow and Data Flow: Concepts of Distributed Programming* (M. Broy, ed.), pp. 399–437. Springer-Verlag, New York, 1985.

[23] J. L. A. van de Snepscheut. *Lecture Notes in Computer Science, vol. 200: Trace Theory and VLSI.* Springer-Verlag, New York, 1985.

[24] N. Soundararajan. "Denotational Semantics of CSP". *Theoretical Computer Science* 33 (1984), pp. 279–304.